REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
05-09-2014		

4. TITLE AND SUBTITLE

Next Generation Satellite Communications: Automated Doppler Shift Compensation of PSK-31 Via Software-Defined Radio

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Lanoue, Matthew James

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Academy
Annapolis, MD 21402

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Trident Scholar Report no. 429 (2014)

12. DISTRIBUTION / AVAILABILITY STATEMENT

This document has been approved for public release; its distribution is UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Software-defined radio (SDR) leverages the processing power of computers to create communications systems that run as flexible software applications, where the radio operating parameters can be set or altered by software. SDR satellite communications systems developed today can run on future hardware platforms and update the applications already running on the satellite. GNU Radio, a framework for creating SDR applications, has recently become capable of developing satellite communications systems. One of the major issues with satellite communications is the Doppler shift experienced as the satellite passes overhead. Demodulating signals affected by Doppler shift requires ground stations with circuits dedicated to track and synchronize with the satellite in order to compensate for the Doppler shift. For the PSK-31 waveform, a terrestrial narrowband form of multi-user amateur radio communications for text and simple data messaging, the amount of Doppler shift exhibited by the satellite would prevent communications using standard receivers. This project implemented PSK-31 in GNU Radio as part of a regenerative satellite repeater. Furthermore, the system estimates and pre-compensates for the Doppler shift generated by an orbiting satellite communicating with a ground station. As a result, the Doppler shift observed at the ground station can be reduced from 10 kHz to less than 20 Hz – a level tolerable by most modern receivers.

15. SUBJECT TERMS
Software-defined radio, Doppler-shift, Satellite communications, PSK-31, GNU Radio

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		68	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

U.S.N.A. --- Trident Scholar project report; no. 429 (2014)

NEXT GENERATION SATELLITE COMMUNICATIONS: AUTOMATED DOPPLER SHIFT COMPENSATION OF PSK-31 VIA SOFTWARD-DEFINED RADIO

by

Midshipman 1/C Matthew J. Lanoue
United States Naval Academy
Annapolis, Maryland

_____
(signature)

Certification of Adviser Approval

Associate Professor Christopher R. Anderson
Electrical and Computer Engineering Department

_____
(signature)

_____
(date)

LCDR Jennie H.G. Wood, USN
Electrical and Computer Engineering Department

_____
(signature)

_____
(date)

Acceptance for the Trident Scholar Committee

Professor Maria J. Schroeder
Associate Director of Midshipman Research

_____
(signature)

_____
(date)

USNA-1531-2

Abstract

Satellite communication systems fall into two broad categories: Amplify and Forward (AF), and Regenerative. AF systems operate as a "bent-pipe" where the information received at the satellite is simply amplified and retransmitted with no alteration of the original signal. In a regenerative repeater, circuitry is designed to demodulate the signal, recover the original information, re-modulate the signal and transmit a new version. Limitations exist in both of these categories: AF systems are unable to compensate for distortion and hardware-defined regenerative repeaters cannot be updated over time.

Software-defined radio (SDR) leverages the processing power of computers to create communications systems that run as flexible software applications, where the radio operating parameters can be set or altered by software. SDR satellite communications systems developed today can run on future hardware platforms and update the applications already running on the satellite. GNU Radio, a framework for creating SDR applications, has recently become capable of developing satellite communications systems.

One of the major issues with satellite communications is the Doppler shift experienced as the satellite passes overhead. Demodulating signals affected by Doppler shift requires ground stations with circuits dedicated to track and synchronize with the satellite in order to compensate for the Doppler shift. For the PSK-31 waveform, a terrestrial narrowband form of multi-user amateur radio communications for text and simple data messaging, the amount of Doppler shift exhibited by the satellite would prevent communications using standard receivers.

This project implemented PSK-31 in GNU Radio as part of a regenerative satellite repeater. Furthermore, the system estimates and pre-compensates for the Doppler shift generated by an orbiting satellite communicating with a ground station. As a result, the Doppler shift observed at the ground station can be reduced from 10 kHz to less than 20 Hz – a level tolerable by most modern receivers.

Keywords

- Software-defined radio
- Doppler-shift
- Satellite communications
- PSK-31
- GNU Radio

Acknowledgements

- Dr. Bruninga for his invaluable knowledge of amateur radio
- Tom Rondeau, Tim O'Shea, Marcus Leech and all the volunteers that helped to answer my questions and work each day to improve the capabilities of GNU Radio
- Dr. Anderson and LCDR Wood for their time and support over the course of the project

Chapter 1: Introduction

1.1 Statement of Objectives

The goal of this project was to design, implement and test the PSK-31 Amateur Radio communication standard on a satellite communications system using software-defined radio (SDR). The SDR will pre-compensate for the Doppler shift experienced by the satellite while in orbit, eliminating the need for ground stations that track the satellite continuously.

1.2 Brief History of Amateur Radio [1]

Amateur radio began in 1831 when Michael Faraday demonstrated the principle of induction. Faraday connected a power source to a wire that he coiled around one end of an iron ring. At the other end of the ring, he coiled a second wire and attached it to an ammeter to measure current. When Faraday connected the power source to the first loop, he discovered that a current was briefly induced in the second loop and demonstrated that electromagnetic energy could be transmitted through a medium. This process was termed mutual inductance and lies at the foundation of wireless communications. A recreation of Faraday's experiment is shown in figure 1.1.

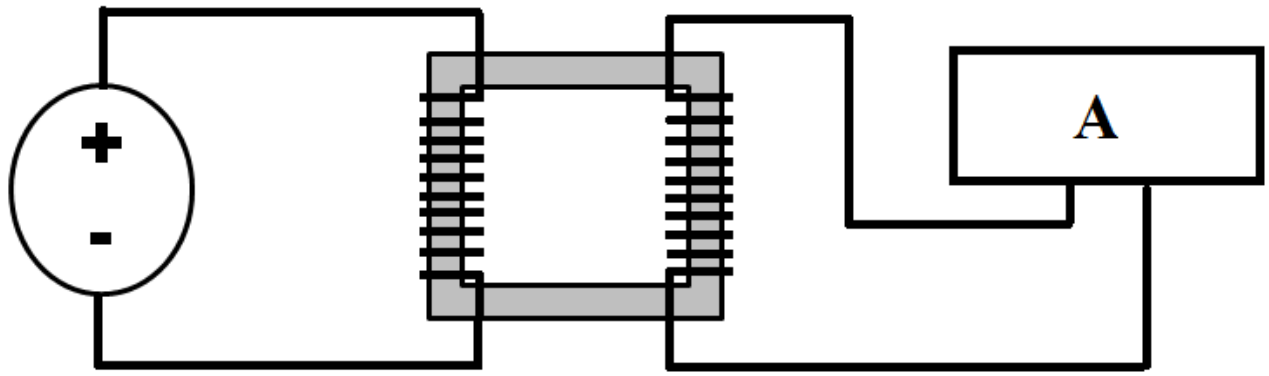

Figure 1.1 Faraday's Induction Experiment

Then, in 1850, James Maxwell derived a set of four equations that relate electricity and magnetism, helping to prove that light is an electromagnetic wave. In 1864, Mahlon Loomis was able to demonstrate wireless transmission of the telegraph over a distance of eighteen miles using two kites as antennas. Later, in 1870, Loomis demonstrated ship-to-ship communications over two miles in the Chesapeake Bay under sponsorship by the United States Navy.

Heinrich Hertz proved Maxwell's equations through experimentation in 1886 with an oscillator and receiver built for waves with wavelengths different from visible light. Guglielmo Marconi expanded upon the work done by his predecessors to send wireless messages across the English Channel in 1899 and then across the Atlantic Ocean in 1901. Soon after, Reginald Fessenden created the world's first radio broadcast of voice and music on December 24, 1906.

The first Amateur Radio club formed in 1909 to continue experimentation and development in

the field of wireless communications. One such development was the superheterodyne principle, which was discovered by amateur radio operator Edwin Armstrong in 1918. By utilizing a local oscillator at a fixed frequency, superheterodyne receivers provide a high degree of frequency agility. This principle is still used today in nearly all communication systems to maximize the use of the frequency spectrum. Figure 1.2 illustrates a typical superheterodyne receiver. After intermediate frequency amplification is completed (IF Amp), additional signal processing can occur to demodulate the signal.

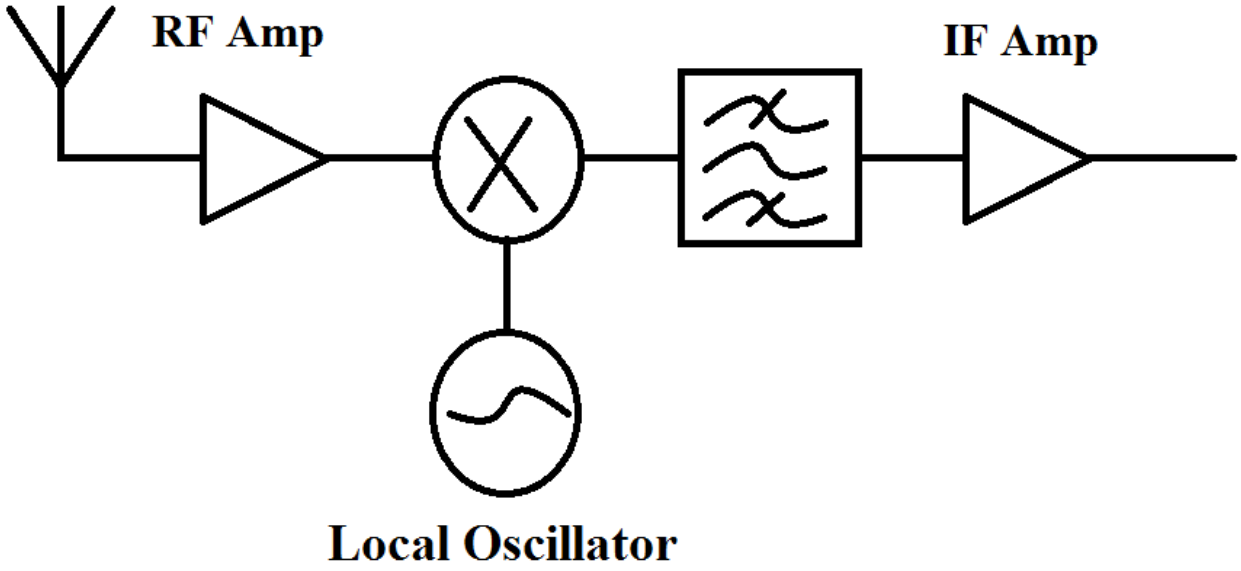

Figure 1.2 Superheterodyne Receiver Block Diagram

In 1957, Sputnik I became the first man-made satellite to orbit the Earth and transmit a wireless signal. Soon after, in 1961, OSCAR-I (Orbital Satellite Carrying Amateur Radio) was launched and amateur radio was brought to outer space. Over 570 amateur radio operators in 28 countries received the Morse code signal "HI-HI" sent by the satellite [2].

Since then, over 100 satellites have been launched solely for use by amateur radio operators. Today, only 20 of those satellites are fully operable. Of those satellites that are still fully functional, only two were made in the USA. The United States Naval Academy has launched eight successful satellite projects created by Midshipmen, but none are fully functional anymore.

1.3 Communications Systems

All communications systems can be described using a few key elements. Fig. 1.3 displays a simple model for a communications system. The information (a picture, some text or a video) is converted into a signal by the transmitter. The transmitter then sends the signal through the channel, which could be a wire or cable or the air. No matter what medium constitutes the channel, the signal is distorted as it passes through the channel. When the signal is picked up by the receiver, it is slightly different from the signal that was transmitted. The differences between the received signal and the original signal depend on the composition of the channel.

Figure 1.3 Simple Communications System Model

The first communications systems utilized analog amplitude and frequency modulation. In analog modulation, the entirety of the information is transmitted over the air. While analog modulation systems were simple to design and create, they were highly susceptible to noise and inefficient in their use of the frequency spectrum. As more customers began using communication systems, operators began looking for a more efficient way to utilize the frequency spectrum – a limited natural resource. The answer was to move from analog to digital modulation.

Unlike analog modulation, the information sent by a user is not perfectly preserved. In a digitally modulated phone call, the user's voice is converted into bits before being transmitted over the air. The quality of the user's voice heard at the receiving end of the phone call is proportional to the number of bits used to represent the voice. While the user's voice can only be approximated by using bits, use of digital modulation proves advantageous because it more effectively utilizes the frequency spectrum and digitally modulated signals are less susceptible to degradation by noise. The trade-off is that digital modulation requires synchronization and tracking, whereas analog modulation does not.

Synchronization is required in digital modulation schemes because the receiver needs to know where bits start and stop. If the receiver is not in synch with the start of each bit, then bit errors will occur. Any bit error will alter information, while significant bit errors will garble or even destroy information. As part of maintaining synchronization, the communications system must track when bits start and stop.

Since digital modulation utilized the frequency spectrum more efficiently, service providers could allow more users to access their networks simultaneously. This led to the development of purely digital communications such as the Internet, cellular phones and Wi-Fi.

1.4 Modulation Schemes

Modulation is the actual process of modifying one or more aspects of a high frequency carrier to convey information as seen in Eqn 1.1.

$$s(t) = A * \cos(2 * \pi * f_c * t + \theta) \qquad \textbf{Equation 1.1}$$

The three major types of modulation include amplitude modulation (changing the value of A), frequency modulation (changing the value of f_c), and phase modulation (changing the value of θ). Examples of these types of modulation are shown in Fig. 1.4.

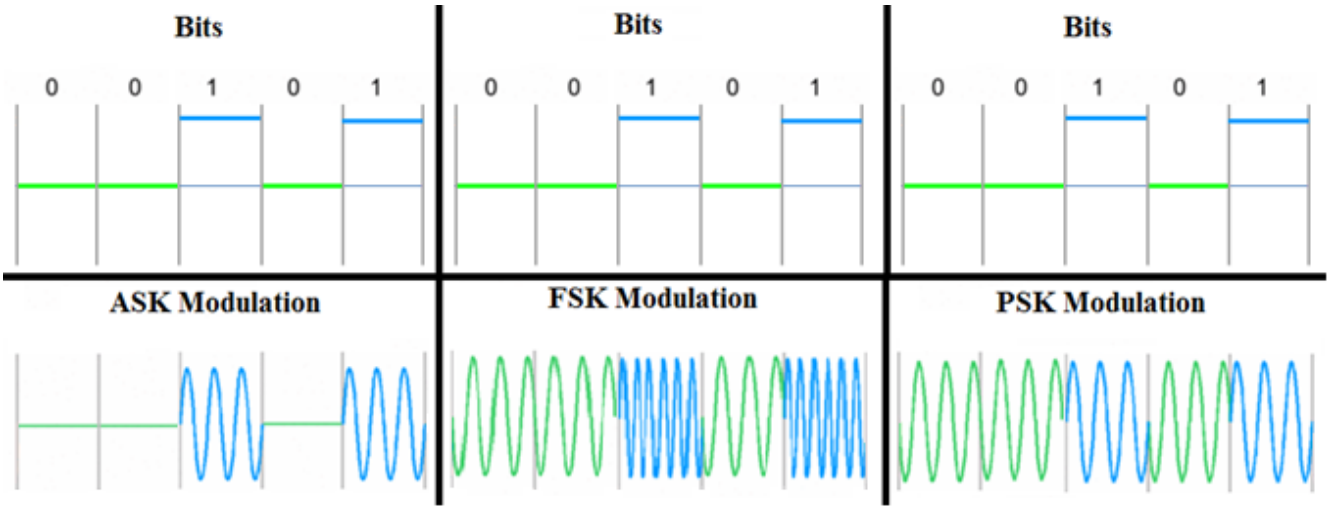

Figure 1.4 Digital Modulation Schemes

Amplitude-shift keying (ASK) conveys information by altering the amplitude of the carrier signal in direct proportion to the information. A high bit, or 1, in an ASK system may correspond to a high amplitude of the carrier signal, while a low bit, or 0, corresponds to a lower amplitude. This is a lot like how the crowd reacts at a football game. When the home team scores, the crows gets really excited and there is a loud noise heard throughout the stadium. When the away team scores, there is some excitement by the visiting fans, but the stadium is nowhere near as loud as when the home team scored. The information in this scenario is the score of the game and the level of noise in the stadium represents the signal.

In frequency-shift keying (FSK), information is conveyed by changing the frequency of the carrier signal in direct proportion to the information. A high bit in an FSK system may correspond to a higher frequency of the carrier signal, while a low bit corresponds to a lower frequency of the carrier signal. This is best represented by a crying baby. If something is done that further upsets the baby, its cries become even higher pitched, but if something is done that pleases the baby, its cries drop to a lower pitch. The baby is trying to convey how upset it is through the pitch, or frequency, of its cries.

Phase-shift keying (PSK) conveys information by altering the phase angle of the signal. Low bits may correspond to the original signal while high bits correspond to the carrier signal shifted 180° out of phase. An example of PSK would be telling students to flip a multi-color button on their desk to answer a question. If the button was blue on one side and gold on the other, the teacher could tell the class to flip to one color or the other in response to the question. In order to understand the students' answers, both the question and the responses need to be known.

After being modulated, the carrier signal is then transmitted across the channel and picked-up by the receiver of another unit. The information is then extracted from the modulated carrier.

1.5 Problems with Satellite Communications

Communicating with a satellite is more difficult than communicating between two ground-level users directly. The signal transmitted by the first user must have enough power to reach the

satellite, which orbits anywhere from 700 km to over 35,000 km above the Earth's surface. Signals are affected by atmospheric conditions such as rain and snow. Just like DirecTV becomes fuzzy in the rain, so too does the rain affect satellite radio communications. An additional issue stems from the fact that the satellite travels at over 17,000 miles per hour in its orbit. For a 700 km orbit, the result is a ten minute window when the satellite is directly overhead and accessible for communications. The short window means that communications must be quick, atmospheric conditions must be favorable and equipment ready in preparation for when the satellite passes overhead or else communications will not occur.

Doppler shift is the change in frequency of a wave detected by an observer when the source of the wave is moving relative to the observer. A common example is how a stationary person hears a change in pitch of an ambulance siren as the ambulance rushes towards and past the observer. The pitch of the siren becomes higher as the ambulance approaches the person and then becomes very low as it passes the observer. Any wireless communications system where the transmitter or receiver is moving will experience Doppler shift – even cellular telephones. The satellite is moving at over 17,000 miles per hour and so the Doppler shift has a significant effect on communications with the satellite; if the Doppler shift is too large relative to the bit rate, synchronization will be lost. Table 1.1 shows the Doppler shift for Wi-Fi at 2.4 GHz for a sources moving at different speeds.

Method of Travel	Speed	Doppler shift
Walking	3 miles/hr	10.74 Hz
Jogging	7 miles/hr	25.05 Hz
Car	65 miles/hr	232.62 Hz
Plane	550 miles/hr	1.97 kHz
Satellite	17,000 miles/hr	60.84 kHz

Table 1.1 Doppler shift for Wi-Fi at 2.4 GHz over a Variety of Speeds

In order to communicate with a moving satellite, either the satellite or ground station must compensate for the Doppler shift. Fig. 1.5 below illustrates Doppler shift as it applies to satellites.

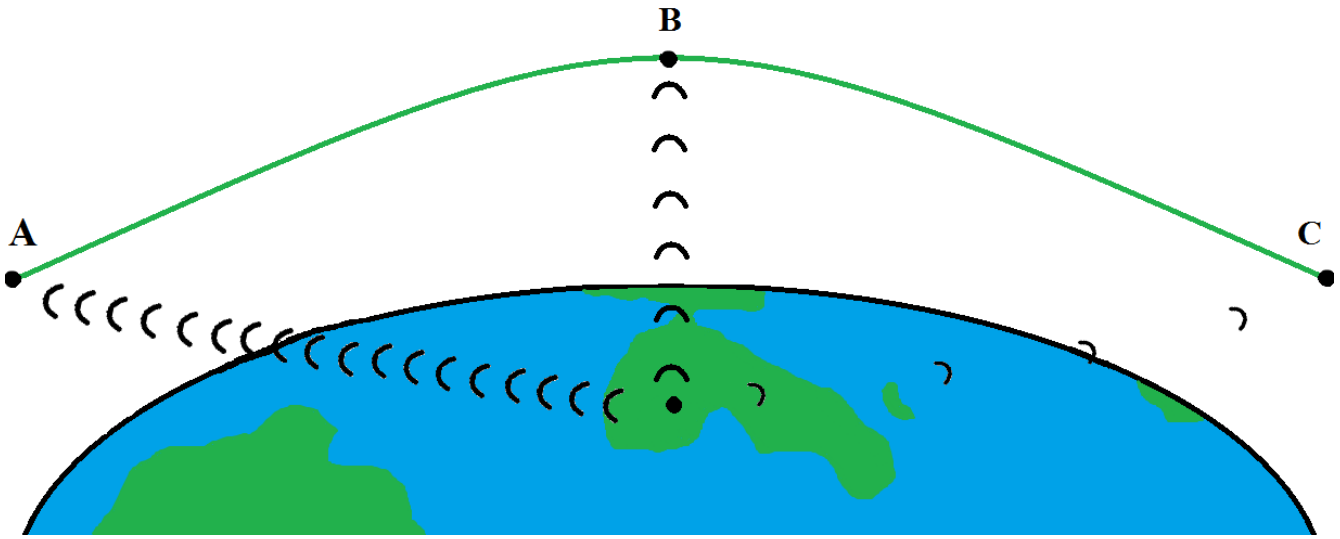

Figure 1.5 Illustration of the Effect of Doppler Shift on Satellite Communications

When the satellite is directly over the ground station at point B, the received frequency at the ground station is equal to the frequency transmitted by the satellite. As the satellite moves towards point C, the received frequency at the ground station is less than the frequency transmitted by the satellite. Between points A and B, the frequency received at the ground station is greater than the frequency transmitted by the satellite.

Chapter 2: PSK-31 Explained

2.1 Definition

PSK-31 is one of many standards that amateur radio operators utilize to communicate with one another. The PSK-31 standard was created by Peter Martinez (call sign G3PLX) in December of 1998 and has become popular for simple text-based communications in the amateur radio community [3].

2.2 Nomenclature

The name PSK-31 is a quick reference to the most important characteristics that comprise the transmitted signal. PSK-31 sends data at 31.25 bits per second. Instead of using 8-bit ASCII representations of characters, a unique character encoding called Varicode is utilized.

2.3 Higher Order PSK

PSK, like all multi-level digital communication schemes, can have two (BPSK), four (QPSK), eight, or more different phases—as long as the number of phases is a power of two. While BPSK uses two phases (each 180° apart), QPSK uses four phases (each 90° apart). Equation 2.1 establishes the relationship between the number of symbols (M) and the number of bits per symbol (N).

$$M = 2^N \qquad \textbf{Equation 2.1}$$

Since the information is transmitted as bits, the extra phases can be utilized to send two bits per symbol instead of the one bit per symbol that sent in BPSK. Fig. 2.2 shows a BPSK and QPSK transmission; each symbol is represented by a different color. To the right of the figure, the constellation diagram illustrating the phases for each modulation scheme is shown.

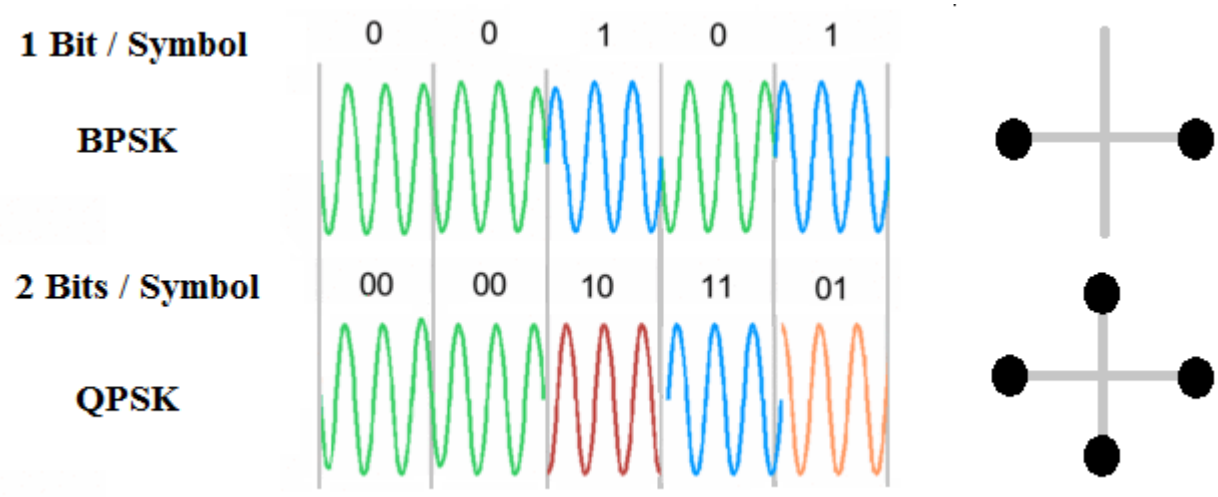

Figure 2.2 Understanding Symbols: BPSK and QPSK

The bandwidth of a PSK transmission (BW) depends on the number of bits per symbol (N) and the symbol rate (R_b) as shown in Eqn. 2.2.

$$BW = \frac{2*R_b}{N} \qquad \textbf{Equation 2.2}$$

For a given data rate (the rate at which bits are sent), a BPSK transmission will have a symbol rate equal to the data rate. A QPSK transmission will have a symbol rate that is half of the data rate and an 8-PSK transmission will have a symbol rate that is a quarter of the data rate. Of the three, the BPSK transmission will have the largest bandwidth and the 8-PSK transmission will have the smallest bandwidth. Fig. 2.3 illustrates the relationship between the number of symbols and the bandwidth of a PSK signal.

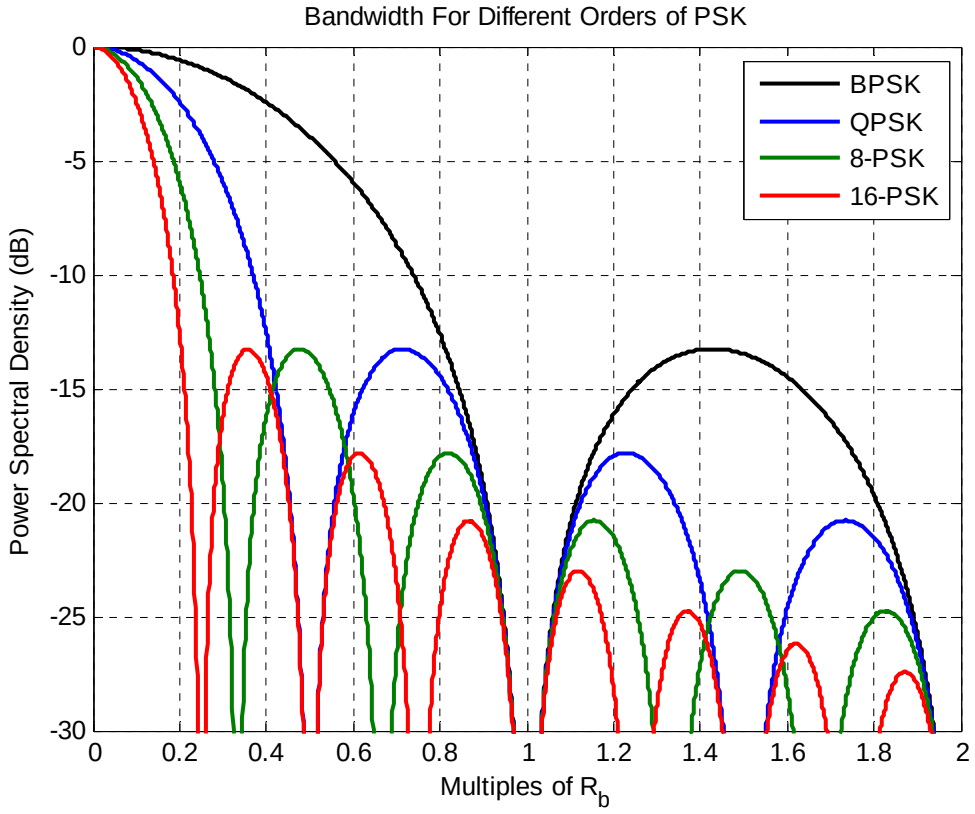

Figure 2.3 Relationship Between Number of Symbols and Bandwidth of PSK Signals

Another way to compare communications standards is to look at the bit error rate (BER). A bit error is counted whenever the received bit is different from the transmitted bit. BER rate is a measure of how many bit errors occurred divided by the number of bits sent. A single bit error marks a distortion in the transmitted information signal, but enough bit errors can garble or ruin the entire signal. For BPSK and QPSK, BER is given by Eqn. 2.3.

$$BER = Q\left(\sqrt{2 * \frac{E_b}{N_0}}\right) \qquad \textbf{Equation 2.3}$$

In Eqn. 2.3, $\frac{E_b}{N_0}$ refers to the energy per bit (E_b) divided by the noise spectral density (N_0). BER is typically plotted against $\frac{E_b}{N_0}$ so that comparisons can be made between different modulation schemes. Fig. 2.4 plots BER against $\frac{E_b}{N_0}$ for BPSK, QPSK, 8-PSK and 16-PSK. For a given BER, 8-PSK and 16-PSK required higher levels of $\frac{E_b}{N_0}$ than BPSK or QPSK.

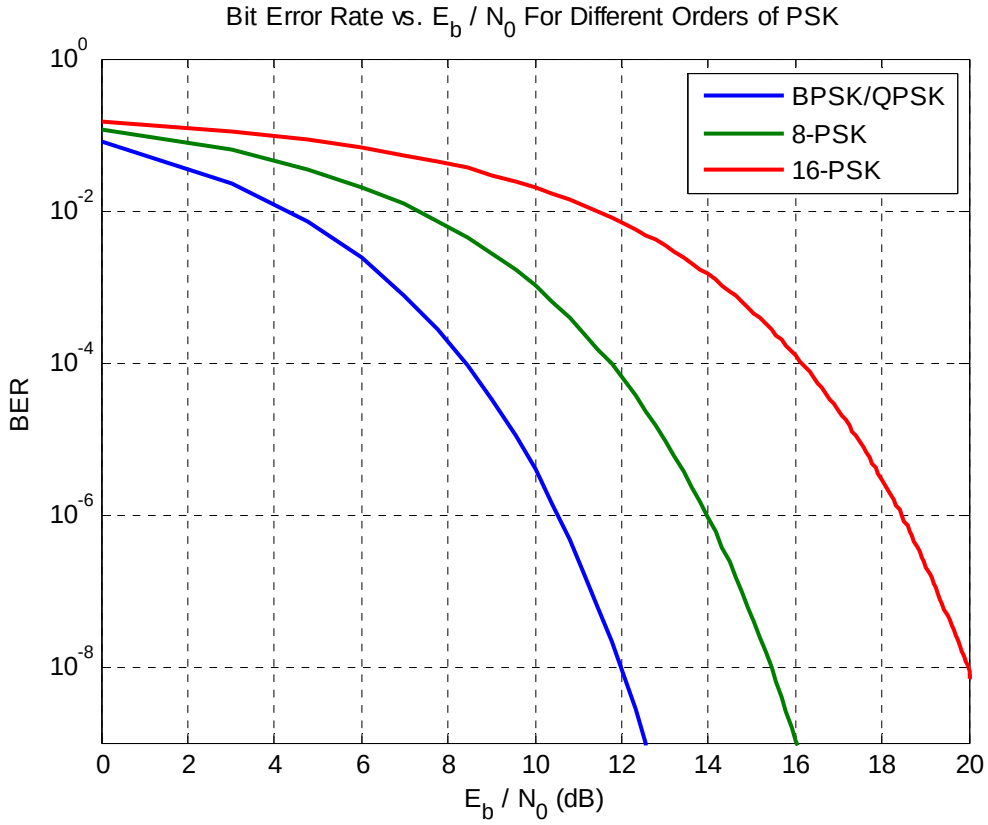

Figure 2.4 Bit Error Rate vs. E_b/N_0 for Different Orders of PSK

Although BPSK and QPSK share the same BER curve, BPSK was utilized for this project because it is the default implementation of PSk-31. Since the satellite is expected to exhibit a large Doppler shift, the bandwidth of the modulation scheme used will small in comparison. By choosing BPSK over QPSK, the satellite is presented with a larger bandwidth to track. Furthermore, a QPSK communications system is more complex than a BPSK communications system.

2.5 Differential PSK

Differential phase-shift keying, or DPSK, is a slightly modified form of PSK. Instead of having distinct phases represent symbols, the presence or absence of a change in phase represents a symbol. Typically, the absence of a phase change from the previous symbol is used to represent a '0' and the presence of a phase change from the previous symbol is used to represent a '1'. Fig. 2.5 shows the difference between a signal represented in BSK and in DBPSK.

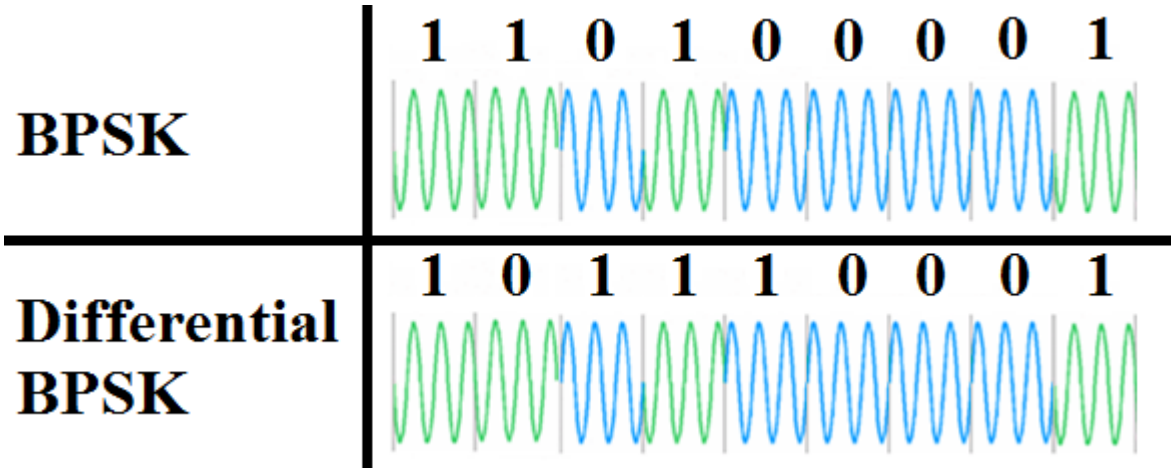

Figure 2.5 Differences Between BPSK and DBPSK

In PSK-31, the presence of a phase change represents a '0' and the absence of a phase change represents a '1'. Since PSK-31 utilizes two symbols, it would be more accurately labeled DBPSK-31.

2.6 Symbol Rate

Assuming that the average user can type at a rate of 50 words per minute, each word is about 6 characters long and each character is 6.5 bits long (the average bit length for Varicode), the user is creating about 32.5 bits per second of data [4]. Since most amateur radio users operate software that runs between 8 kHz and 44.1 kHz (common computer sound card operating frequencies), the data rate of 31.25 Hz was picked for use in PSK-31. This data rate would divide into 8 kHz evenly (8000/31.25 = 256).

Therefore, the 31 in PSK-31 refers to the data rate used – 31.25 bits per second. Since binary phase shift keying is being utilized, each symbol is transmitted for 32 milliseconds. Eqn. 2.4 establishes the relationship between the symbol period, T_s, and the symbol rate, R_b.

$$T_s = \frac{1}{R_b} \qquad \textbf{Equation 2.4}$$

2.7 Varicode Encoding [5]

While most computers use ASCII tables to convert text into data, PSK31 uses a different system called Varicode. The difference is that, unlike ACII where each character is represented by the same number of bits (8), characters are represented by different numbers of bits in Varicode – hence the name "Variable Code". To distinguish between the end of one character and the start of another, there is a gap containing two or more zeroes between each character. As a result, each character starts and ends with a one and cannot contain two consecutive zeroes within the Varicode sequence. The shortest possible Varicode bit sequence is '1' which represents the space character. Table 2.1 displays the Varicode representation for various characters.

Varicode	Character	Varicode	Character	Varicode	Character	Varicode	Character
1	Space	1011	a	11011	l	1101011	w
10110111	0	1011111	b	111011	m	11011111	x
10111101	1	101111	c	1111	n	1011101	y
11101101	2	101101	d	111	o	111010101	z
11111111	3	11	e	111111	p	1111101	A
101110111	4	111101	f	110111111	q	1010101101	Z
101011011	5	1011011	g	10101	r	111111111	!
101101011	6	101011	h	10111	s	111110101	#
110101101	7	1101	i	101	t	111011011	$
110101011	8	111101011	j	110111	u	1010111101	@
110110111	9	10111111	k	1111011	v	1011010101	%

Table 2.1 Varicode Representations for Select Characters

Varicode was designed such that the most common characters have the shortest bit sequences. The longest bit sequences are 10 bits long. Table 2.2 illustrates the number of characters by length of Varicode bit sequence. Within a typical English text, the average number of bits per character is 6.5 compared to the 8 bits per character used in ASCII. This average is not based solely on the average length of bit sequence – which would be approximately 8.27 bits per character – but also on the frequency with which each character appears in a typical English text.

Length of Bit Sequence	1	2	3	4	5	6	7	8	9	10
Number of Characters	1	1	2	3	5	8	13	21	34	40

Table 2.2 Number of Characters by Length of Varicode Bit Sequence

2.8 Generation of PSK31

Normally a phase-shift keying system requires the receiver to have a copy of the signal's phase before it was modulated. Without this copy, the receiver cannot tell which phase represents one and which represents zero. Thus, PSK requires precise synchronization between the transmitter and receiver which must be maintained throughout the transmission of data. An ordinary PSK system is difficult to create even when the transmitter and receiver are both stationary and much more complicated to maintain when the transmitter or receiver start moving.

In order to synchronize at the receiver, there must not be a long series of bits without a phase reversal. By choosing a phase reversal to represent a '0' bit and no change in phase to represent a '1' bit, there will never be more than 10 bits without a phase reversal. As long as each transmission starts with a long enough string of zeroes, the receiver can use the phase reversals to determine where one bit ends and the next begins. For a PSK-31 transmission, a "preamble" of 81 phase reversals precedes any information being sent [6].

Following the preamble, a digital one is indicated by no phase shift from the previous bit, while a digital zero is marked by a 180° phase shift from the previous bit. For example, the character "U" in Varicode is represented by the bit sequence 101010111. Fig. 2.6 shows the last four bits of the preamble followed by the character "U". Notice that two zeroes are transmitted before and after the bit sequence for "U" to separate the character from any preceding or following characters. The bits above the graph are the information that is being transmitted, while the graph below illustrates what voltages levels are actually transmitted.

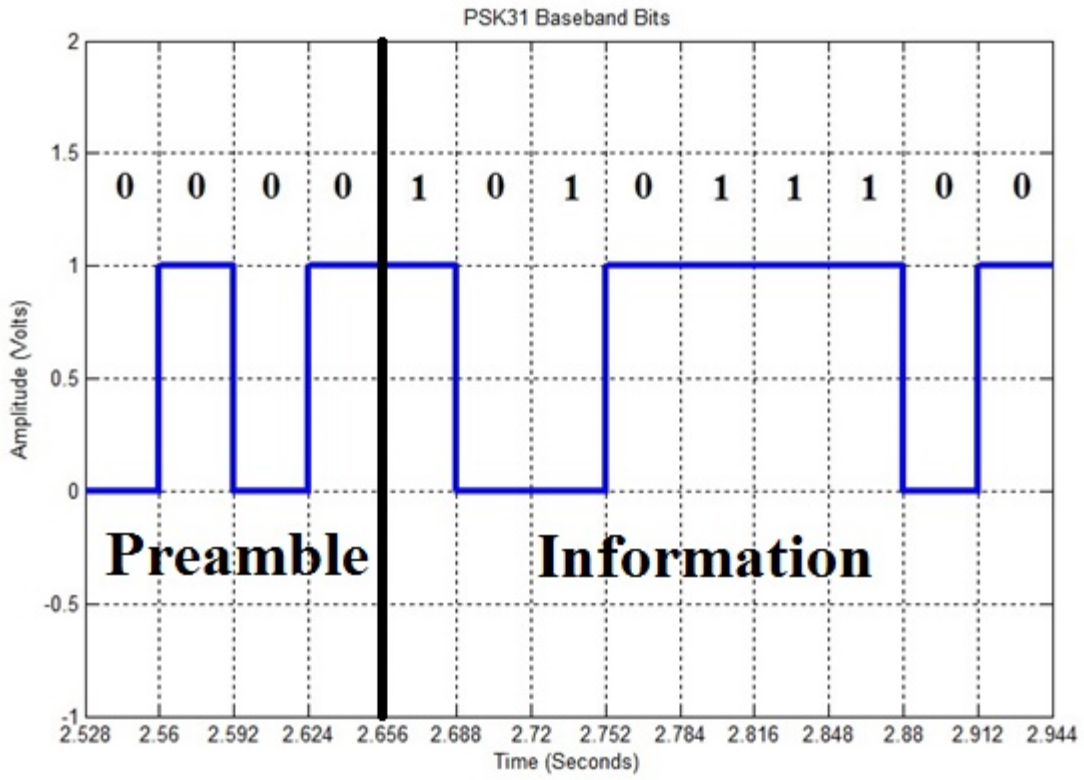

Figure 2.6 Digital Representation of "U"

PSK-31 specifies that the bits are sent through a low-pass filter with a cutoff frequency of 15 Hz. Without this filter, extraneous sidelobes would be seen at integer multiples as seen in the top frequency spectrum in fig. 2.7 and interfere with adjacent users. The low-pass filter removes all sidelobes beyond 15 Hz. This limits the PSK31 transmission to 31.25 Hz of bandwidth and prevents one user from interfering with other users operating at different frequencies. Fig. 2.7 shows the frequency spectrum of the baseband bits before (top) and after the low-pass filter (bottom). While the pulse-shaping low-pass filter does not completely remove the lobes, it does reduce the power at these spikes by about 20 dB. At these levels, the shifted copies would be drowned out by noise in the channel.

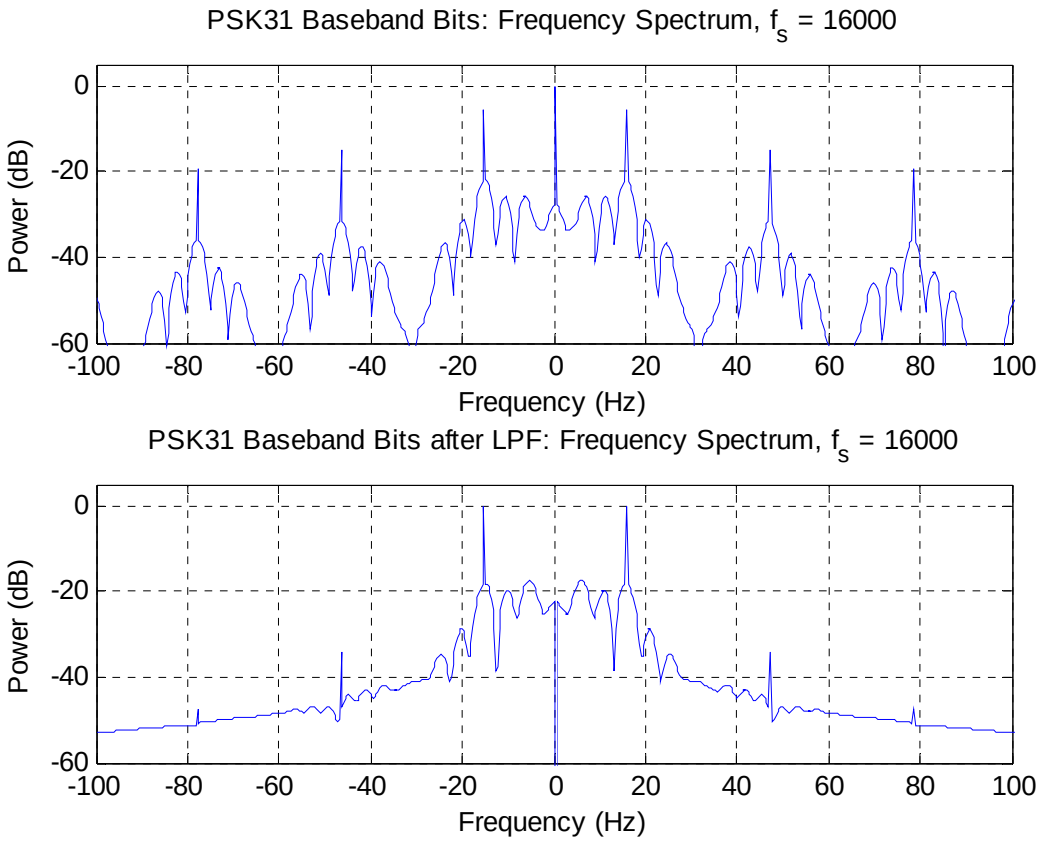

Figure 2.7 Frequency Spectrum of Baseband Bits before and after Low-pass Filtering

Typical frequency bands in the very high frequency (VHF) range used for PSK-31 communications are listed in table 2.3. To maximize use of the frequency spectrum, the superheterodyne principal is utilized by amateur radio operators. Signals are multiplied by a desired intermediate frequency called the subcarrier frequency and then multiplied again by the desired carrier frequency. The advantage of the superheterodyne principal is that it allows amateur radio operators to use common hardware, while still utilizing the full frequency spectrum.

Frequency Band	Frequency
6 m	50.29 MHz
2 m	144.144 MHz
1.25 m	222.07 MHz
0.70 m	432.2 MHz
0.33 m	909 MHz

Table 2.3 Common Frequency Bands Utilized for PSK-31 Communications [7]

Once the signal has run through the low-pass filter, it is next multiplied by a subcarrier to bring the signal to a desired intermediate frequency. Fig. 2.8 shows a PSK31 signal in the time domain before (top) and after (bottom) it is multiplied by a 1 kHz subcarrier wave. One key feature to note is that sharp transitions between bits seen in fig. 2.6 have been smoothed out by the low-pass filter. The last eight bits of the preamble are shown, followed by the character “U”.

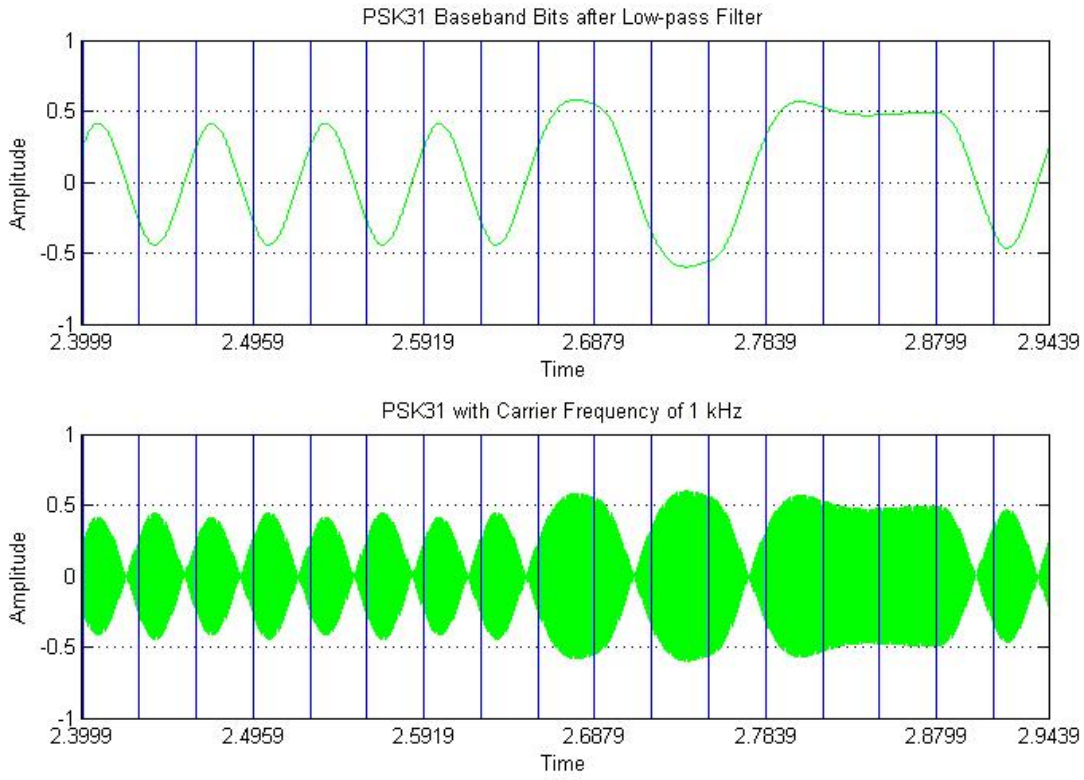

Figure 2.8 Time Domain View of PSK31 before (top) and after (bottom) multiplication by a 1 kHz Subcarrier

Following the information section of the PSK31 signal is a "tail" that consists of 750 milliseconds of the subcarrier without modulation. The purpose of the tail section is to notify the receiver that the transmission is ending and prevent the creation of garbage characters that were not intended to be sent [8]. When fully combined, a PSK31 signal has three pieces: the preamble, the information and the tail.

Chapter 3: GNU Radio and the USRP

3.1 Software-defined Radio

A software-defined radio (SDR) is a communications system that utilizes software instead of hardware components to carry out signal processing. Modern smartphones can be thought of as a prevalent example of SDR. Instead of designing circuits for each function on the phone (text messaging, e-mail, internet browsing, etc.), each function is created in software which is then selected by the user and executed by the hardware – the computer that serves as the "brain" of the smartphone. Since the applications were written in software, they can be run on a variety of cellular phones and they can be updated remotely. Changing an application does not require the user to bring their phone into a service center so that hardware can be replaced to fix a bug in the e-mail application. The upgrade paths for software and hardware are independent.

For satellite communications, SDR has many of the same benefits as it does for smartphones. Applications – communications systems – can be run on a variety of hardware platforms. As newer generations of hardware are developed, SDR applications can be improved in a few lines of code to take full advantage of the advances in hardware. Additionally, SDR systems can be remotely updated once deployed in orbit. The satellite can be updated to counter an error found in the software or to enhance performance. Currently, satellites that encounter a significant bug must be recaptured and fixed manually by astronauts or, more likely, they are disabled and abandoned.

As of the time that this paper is being written, only one example of SDR can be found in outer space, which is a USRP operated by NASA aboard the International Space Station (ISS) [9]. No known examples of true SDR on board a communications satellite could be found (8 May 2014).

3.2 GNU Radio

GNU Radio is an open-source framework for creating software-defined radio applications. The heart of any GNU Radio application is a flowgraph – a system of functional blocks that perform operations on a stream of data. The flowgraph can be created visually in the GNU Radio Companion (GRC) or textually in a Python file. Fig. 3.1 shows a basic flowgraph created using GRC.

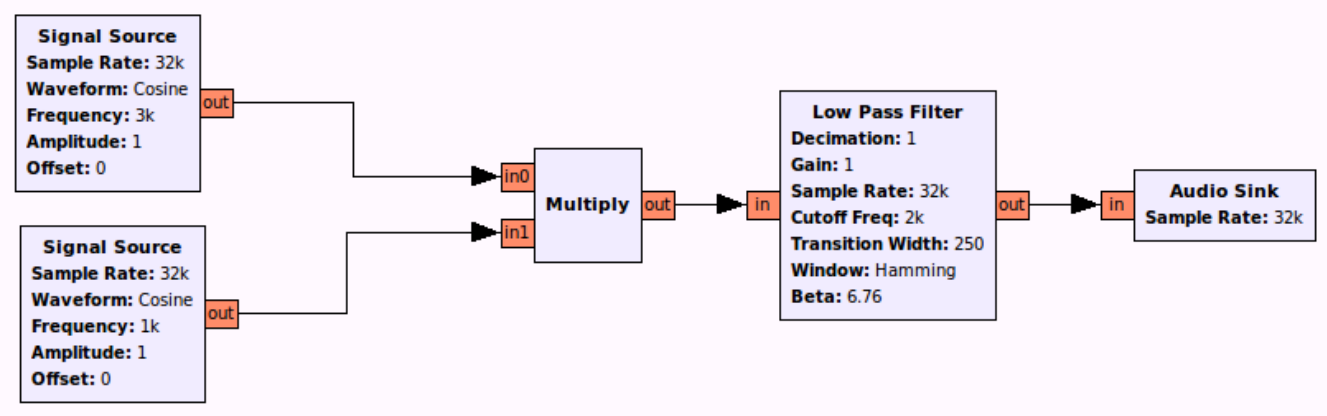

Figure 3.1 Example Flowgraph in GNU Radio

Some blocks are designed to perform a single function, while others combine multiple functions together in a hierarchical block. Each flowgraph must start with a source block and end with a sink block. In Fig. 3.1, there are two source blocks, each creating a cosine wave at a different frequency, and one sink block – an audio sink that will play the resulting waveform over the computer's speakers. Every block requires a specific format for input data and specifies an output format. Connections between blocks can only be made when the output data format of the first block matches the input data format of the second block. For example, the multiply block in Fig. 3.1 outputs a stream of floats and the low pass filter block requires a stream of floats. Since the output type of the multiply block matches the input type of the low pass filter block, a connection could – and was – made between them. Table 3.1 lists the data types available in GNU Radio and the corresponding colors that will appear on the input and output hubs of each block. Vectors of a certain data type are denoted by darkening the color for that data type. For example, a vector of 2048 complex numbers will be denoted by dark blue.

Data Type	Float	Complex	Integer	Unsigned Character	Short Integer
Color	Orange	Blue	Green	Purple	Yellow

Table 3.1 List of the Color Codes by Data Type Available in GNU Radio

Users are provided with a multitude of blocks created by contributors to the GNU Radio community, but they can also create their own blocks in C++ or Python. Table 3.2 lists the major types of blocks available within GNU Radio.

Block Type	Description
Source	These blocks are used to bring data into GNU Radio or create data on which the flow-graph will operate. Waveform generators, audio sources and the USRP source block fall into this category.
Sink	The destination for the data in the flowgraph. Common sink blocks include: file sink, WAV sink, audio sink or USRP sink.
Synchronous	Most blocks within the flowgraph are synchronous: the same number of samples that enter the block will leave the block. Examples include filters, math operators and phase-locked loops.
Decimating	These blocks will destroy samples. The rational resampler and fractional resampler blocks are both decimating blocks.
Interpolating	These blocks will output more samples than they receive. The repeat block is an interpolating block.
Hierarchical	A single block created by combining multiple blocks that are typically used together for a certain purpose. Modulator and demodulator blocks for ASK and FSK systems are hierarchical blocks composed of the functions required to create typical modulators and demodulators.

Table 3.2 List of the Major Block Types Available in GNU Radio

A key feature of GNU Radio is that the software is not designed to work with any specific piece of hardware. While blocks exist to allow SDR applications to run on specific hardware platforms, GNU Radio does not limit users to any one line of hardware. In fact, GNU Radio can operate in simulation mode without any hardware at all.

3.3 Universal Software Radio Peripheral

The Universal Software Radio Peripheral (USRP) is a hardware platform developed specifically for running SDR applications. Two major families of USRP products exist, the basic line and the embedded line. Basic USRP products require a connection to a desktop or laptop PC to develop and run SDR applications. Embedded processor USRP products ship capable of running a version of Linux and can run as a standalone unit. For this project, the USRP E100, from the embedded line of products was chosen to emulate the standalone system on board the satellite.

To interface with GNU Radio and other SDR frameworks, the USRP utilizes the USRP hardware driver (UHD). Within GNU Radio, there is a source and sink block that allows flowgraphs to run on the USRP. Signal processing occurs within the computer or embedded processor, and the UHD sends the results to the appropriate channel in the USRP. While the signal processing capabilities of the USRP are dictated by the embedded processor, the bandwidth of signals that the device can physically create is controlled by the daughterboard currently installed in the USRP as well as the data transfer rate between the processor and USRP.

Chapter 4: Doppler Shift Estimation

4.1 Definition

Doppler shift is the change in frequency of a wave detected by an observer when the source of the wave is moving relative to the observer, as given by Eqn 4.1. In Eqn 4.1, f_d is the Doppler shift expressed in Hertz, v is the velocity at which the source is moving, $\boldsymbol{\lambda}$ is the wavelength of the signal and $\boldsymbol{\theta}$ is the angle between the source and the observer.

$$f_d = \frac{v}{\lambda} \times \cos(\boldsymbol{\theta}) \qquad \textbf{Equation 4.1}$$

A common example of Doppler shift is how a stationary person hears a change in pitch of an ambulance siren as the ambulance rushes towards and past the observer. The pitch of the siren becomes higher as the ambulance approaches the person and then becomes very low as it passes the observer. In the case of satellite communications, the satellite is moving at over 17,000 miles per hour and the Doppler shift has a significant effect on communications with the satellite. For a satellite in Low Earth Orbit (LEO), the maximum Doppler shift expected is about 12 kHz [10].

If the Doppler shift is too great with respect to the bandwidth of the signal and the change in frequency is too large for the satellite to track, synchronization will be lost. Doppler shift is responsible for causing errors in the signal which can obscure or alter the message that is received. In order to communicate with a moving satellite, either the satellite or ground station must track the Doppler shift.

4.2 Determining the Event Horizon

In order to create a family of Doppler shift curves that represent the possible ways for the satellite to pass over a ground station, the event horizon needed to be calculated. The event horizon is the maximum possible distance that a ground station could communicate with the satellite. Assuming that the Earth is roughly spherical and that the ground station is operating at sea level, the event horizon can be calculated with trigonometry. Figure 4.1 illustrates the scenario with the event horizon painted orange, the Earth painted blue and the trajectory of the satellite painted green.

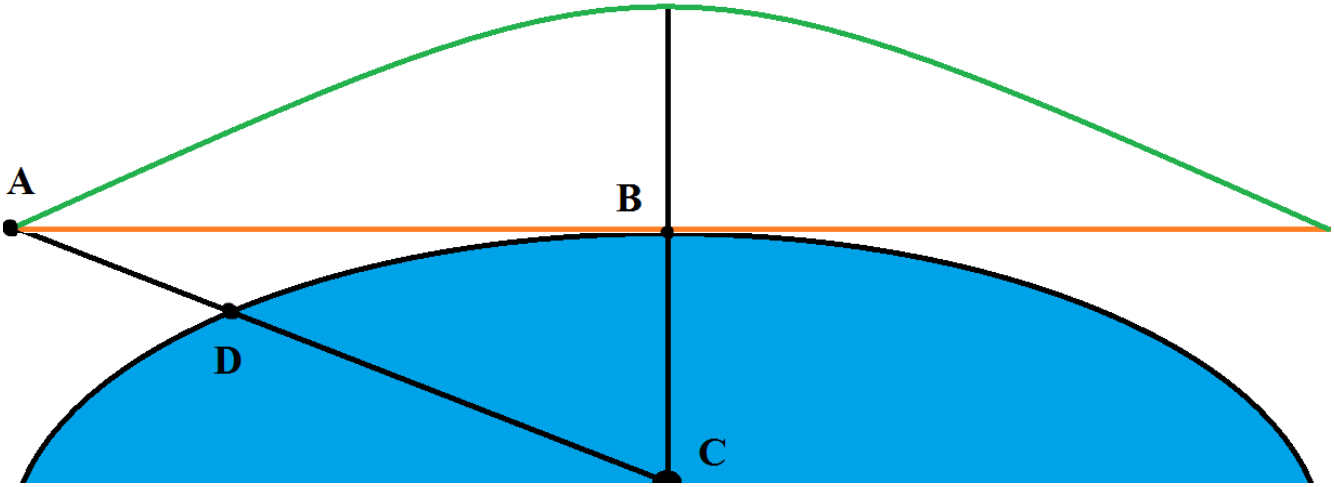

Figure 4.1 Geometry Used to Determine Radius of the Event Horizon

The radius of the event horizon is the distance between points A and B. The length of segments BC and CD is the radius of the Earth and the length of segment AD is the height above the Earth at which the satellite orbits. Utilizing the Pythagorean Theorem yields Eqn. 4.2.

$$AB = \sqrt[2]{AC^2 - BC^2} \qquad \textbf{Equation 4.2}$$

Knowing that the radius of the Earth is approximately 6371 km and that the satellite will orbit at an average height above the Earth of 418.5 km, the radius of the event horizon is found to be 2346.84 km. This approximation of the event horizon assumes that the ground station is operating at mean sea level. To account for a station that was as far from the center of the Earth as possible, the geometry was redone. Fig. 4.2 shows the geometry for a ground station operating at the summit of Mt. Chimborazo, the point furthest from the center of the Earth [11].

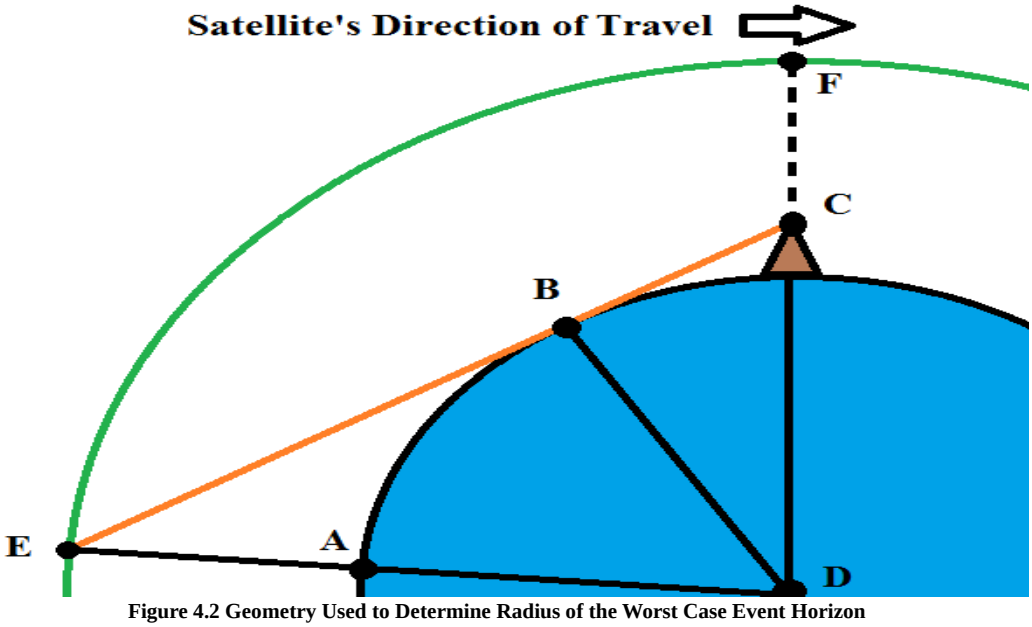

Figure 4.2 Geometry Used to Determine Radius of the Worst Case Event Horizon

For this project, the event horizon was rounded up to 2500 km to account for ground stations that operate slightly above sea level.

4.3 Data Collection

To simulate the pass of a LEO satellite, Doppler shift data was collected from multiple passes of the ISS to estimate the range of expected Doppler shift frequencies for a satellite pass in LEO. The data was used to graph a family of curves that displays the effect of the type of overhead pass on the expected Doppler shift. Direct, overhead passes experience the most rapid Doppler shift, while passes that are more tangential experience the most gradual Doppler shift. Fig 4.3 shows the Doppler curves plotted for a direct, overhead pass (left graph) and a tangential pass (right graph) of the ISS.

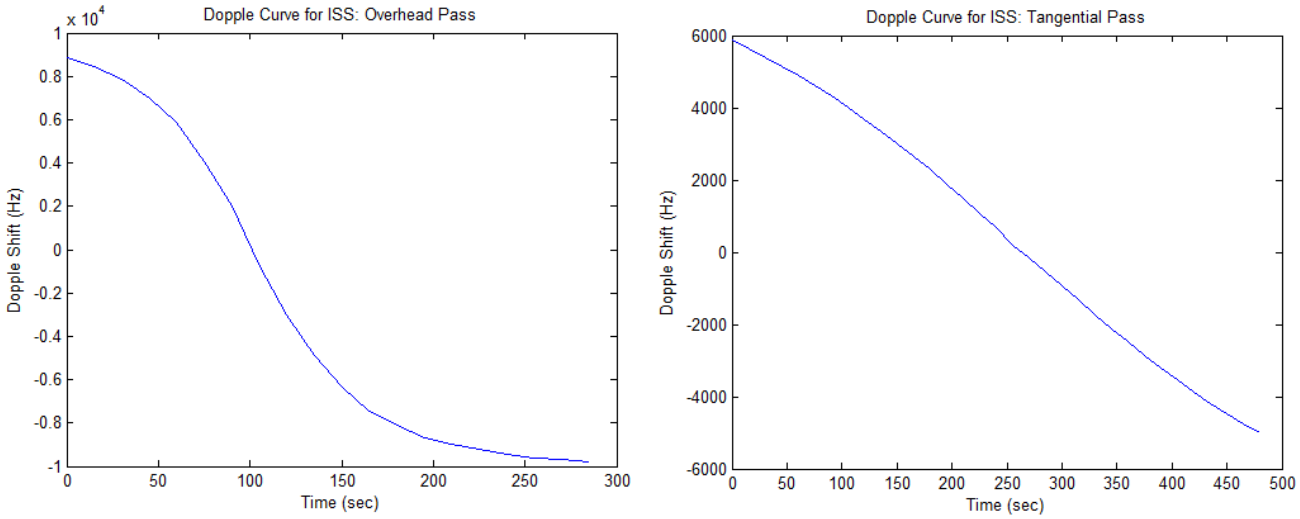

Figure 4.3 Doppler Shift Seen on International Space Station for Different Overhead Passes

4.4 Doppler Curve Generator

The Doppler shift seen at the satellite is dependent upon the relationship between the ground station and the satellite. To generate the Doppler curves, a few simplifying assumptions are made. First, the Earth is assumed to be a perfect sphere over the span of the event horizon. Second, the pass of the satellite is assumed to be at a constant height above the Earth. These two assumptions allow the satellite passes to be represented in a Cartesian coordinate system with the ground station as the origin.

Since the altitude (z-coordinate) is assumed to be constant across the orbit, the pass of the satellite can be simplified into a two-dimensional problem. Given the two points on a planar circle that represents where the satellite passes through the event horizon, Doppler shift can be calculated. For this Doppler Curve Generator, the user selects a point where the satellite pass begins along a vertical line and another point along a second vertical line where the pass ends. The simulation then generates what the overhead pass looks like in the x-y plane.

Using information readily available about the I.S.S., the station can be represented in Cartesian coordinates. The Cartesian coordinates are converted into spherical coordinates so that the change in relative velocity of the satellite with respect to the ground station can be calculated. The change in distance between the I.S.S. and the user (the value of rho), divided by time between samples will give a step approximation of the relative velocity between the user and the satellite. This can then be plugged into eqn. 4.1 to calculate Doppler shift. Fig. 4.4 shows the Doppler curve generated for a tangential pass and fig. 4.5 shows the Doppler curve generated for an overhead pass. Code for the Doppler curve generator can be found in appendix e.

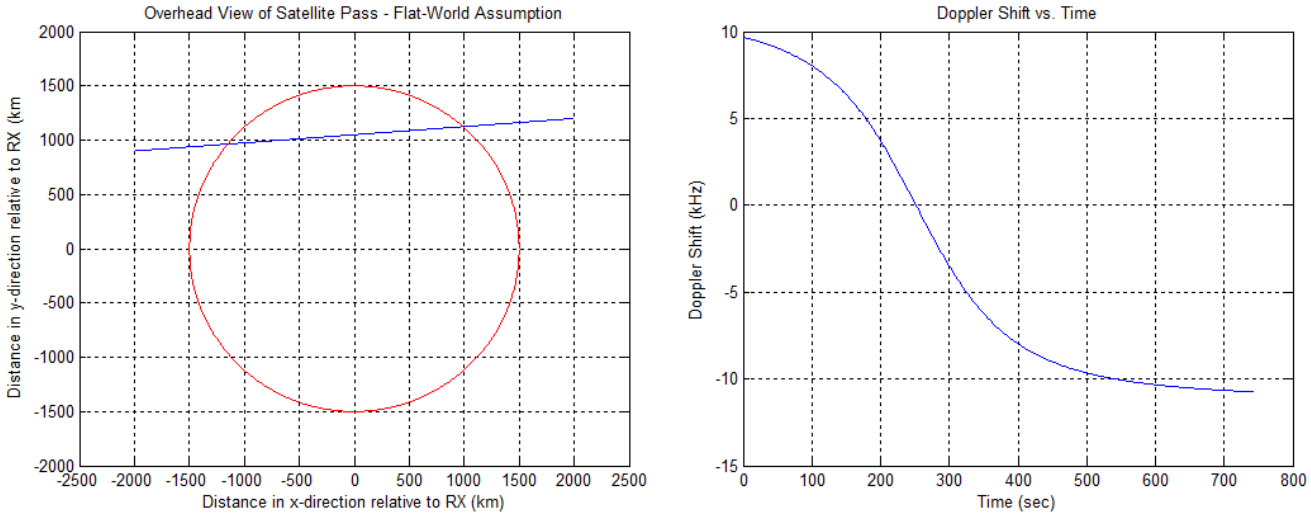

Figure 4.4 Doppler Curve Generated for a Tangential Pass

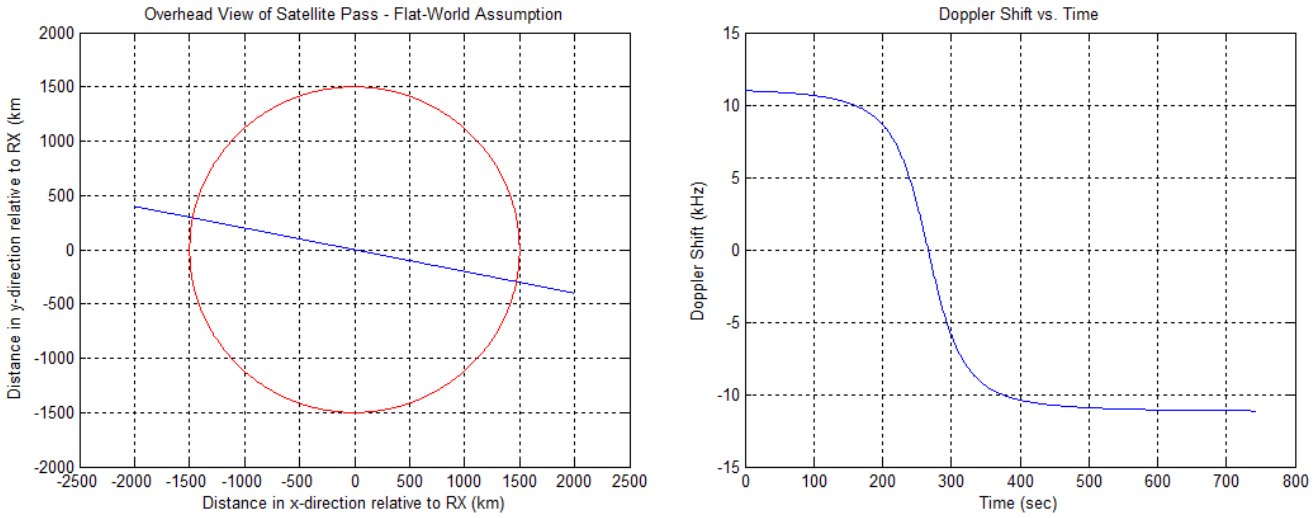

Figure 4.5 Doppler Curve Generated for a Direct, Overhead Pass

Chapter 5: PSK31 Transmitter

5.1 Simulation

MATLAB was used to prototype and evaluate the performance of a PSK31 transmitter due to familiarity with the software, ease of use and data manipulation tools. Once the transmitter was verified to work in MATLAB, the code served as a template for creating the transmitter in GNU Radio. Code for the MATLAB transmitter can be found in the appendix. Fig. 5.1 illustrates the block diagram used to create the transmitter.

Figure 5.1 Block Diagram for PSK31 Transmitter

The first step in creating the transmitter was to take a message input by the user and convert the characters into the corresponding Varicode representation. A MATLAB function looks up each individual character that the user inputs and stores the result in a vector (*code*). After each character, the sequence "0 0" is added to indicate that one character has ended and the next will begin. A preamble of 81 bits worth of zeroes is sent before the first character in the message. This preamble allows the receiver time to synchronize with the signal and helps prevent data loss.

Next, another vector (*base*) is generated with an alternating "1 0" pattern for the length of the *code* vector. This vector represents the phase of the PSK31 signal. Since transmitting a series of zeroes in PSK31 is actually transmitting a series of phase reversals, the *base* vector is what would be transmitted if the user did not send any data or the message string was empty.

The *code* and *base* vectors are combined into another vector called *netshift*. The *netshift* vector accounts for the bits in the *code* vector and varies the phase from the *base* vector appropriately. If the current bit being transmitted is a '1', then the phase of the *netshift* vector will be the same as it was for the last bit. Otherwise, if the current bit being transmitted is a '0', then the phase of the *netshift* vector will be the opposite of what it was for the last bit. Table 5.1 shows the vectors *code*, *base* and *netshift* for the message "14".

code	0	0	1	0	1	1	1	1	0	1	0	0	1	0	1	1	1	0	1	1	1	0	0
base	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
netshift	0	1	1	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	1	1	1	0	1

Table 5.1 Analyses of MATLAB Vectors for the Message "14"

The next step was to expand the *netshift* vector by resampling. Since the bandwidth of a PSK-31 signal is 31.25 Hz, the signal – in the *netshift* vector – is sampled at a rate of 31.25 samples per second. In order to share the frequency spectrum with other users, the signal must be multiplied

with a subcarrier signal at an open slot in the frequency spectrum. The Nyquist equation states that the sampling rate must be at least twice that of the maximum frequency observed in the signal. Since the subcarrier frequency could be up to 8 kHz, the resulting sampling rate would then need to be at least 16 kHz. In MATLAB, this operation entailed copying each item in *netshift* 512 times to up-sample from 31.25 Hz to 16 kHz and storing the result in a vector called *PSK31_base*. Fig. 5.2 shows a MATLAB plot of *PSK31_base* at a sampling rate of 16 kHz.

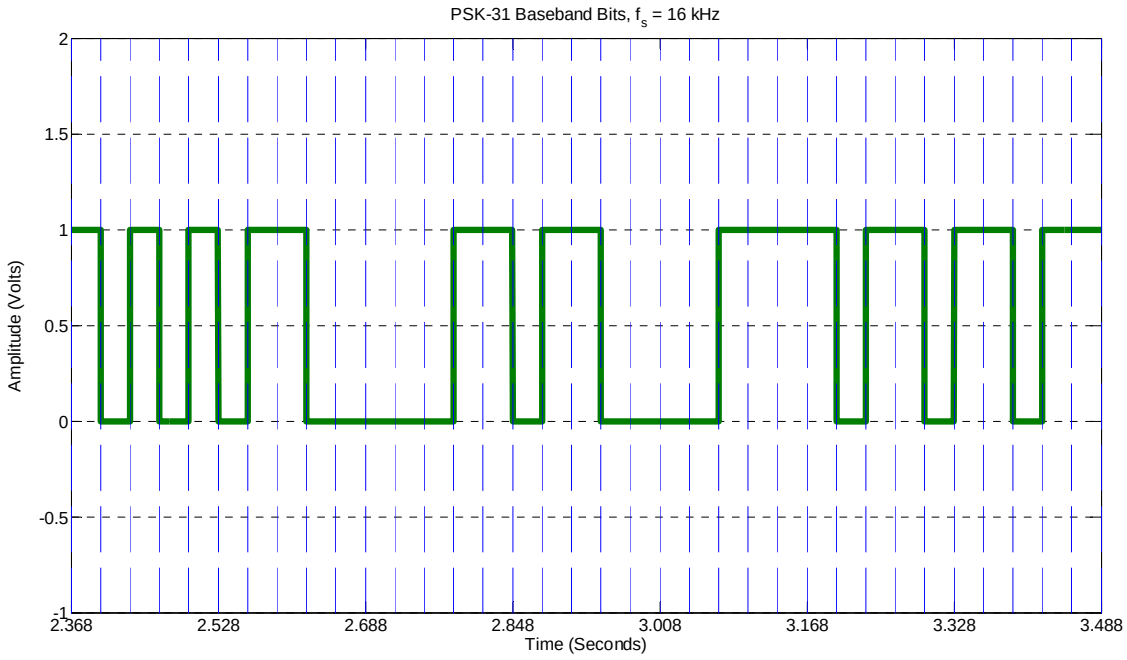

Figure 5.2 MATLAB Plot of "14" Represented in Varicode Bits at a Sampling Rate of 16 kHz

After up-sampling, *PSK31_base* was sent through a low pass filter with a cutoff frequency of 15 Hz and stored in *PSK31_filtered*. Besides being a required step in the creation of a PSK-31 signal, the low pass filter rounds the sharp phase transitions seen in the baseband bit sequence. Fig. 5.3 shows a MATLAB plot of the same bit sequence for "14" after it has been sent through the low pass filter.

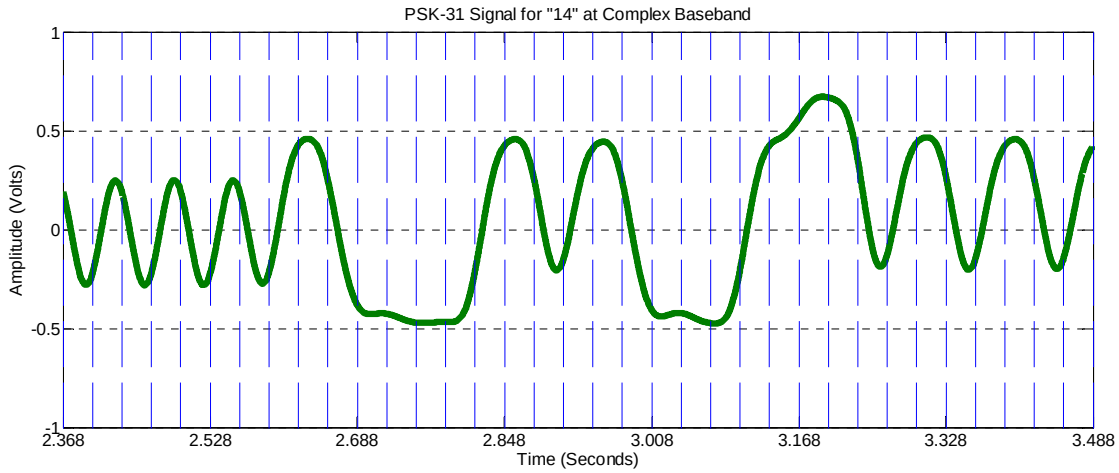

Figure 5.3 MATLAB Plot of "14" After the Bit Sequence was sent through a Low Pass Filter

Then, *PSK31_filtered* is multiplied by a cosine wave at the desired subcarrier frequency (1 kHz was used for testing) and 750 milliseconds of unmodulated subcarrier are added at the end of the signal. Fig. 5.4 shows the result of the subcarrier multiplication. The low pass filter creates a small time delay that causes the blue dashed lines that represent the bit period to misalign with the signal by a small fraction.

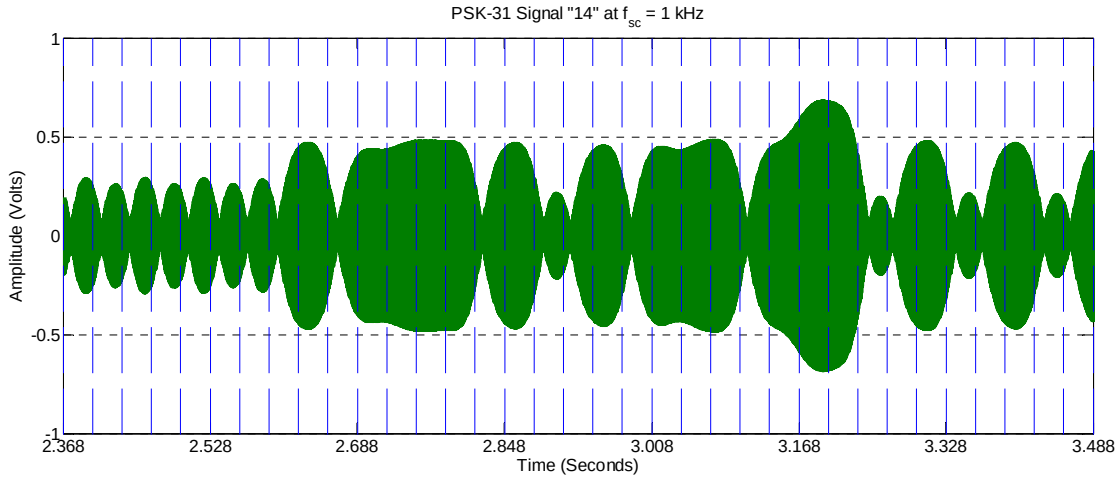

Figure 5.4 MATLAB Plot of PSK-31 Signal for "14" at a Subcarrier Frequency of 1 kHz

One final step was necessary in MATLAB because to eliminate shifted copies of the signal found during testing. A band pass filter was added after the subcarrier multiplication that stretched from 985 Hz to 1015 Hz. Following the band pass filter, the signal was saved in a .WAV file to facilitate testing with Digipan and GNU Radio.

5.2 Hardware Recreation

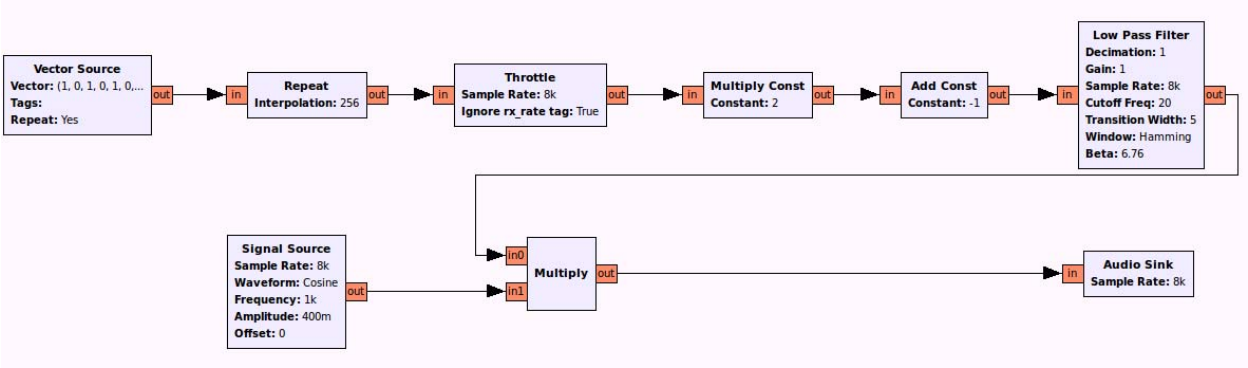

Figure 5.5 GNU Radio Companion Flowgraph for PSK-31 Transmitter

The PSK31 transmitter was constructed in GNU Radio using the code generated in MATLAB as a template. The complete GNU Radio flowgraph for the transmitter is shown in Fig. 5.5. Instead of taking a message from the user, the transmitter requires that the code from the *netshift* vector be directly input into a "vector source" block. This block begins the GNU Radio implementation of the PSK31 transmitter.

The next step is to run the output of the "vector source" block through a "rational re-sampler" block with a resampling rate of 256. This block takes each sample from the source block and creates 255 samples for a total of 256 samples. The audio output of the USRP E100 is limited to sampling rates of 8 kHz or less (instead of the 16 kHz used in the MATLAB simulation). Since the bit rate for PSK-31 is 31.25 Hz, up-sampling by 256 results in a sampling rate of 8 kHz (31.25*256).

Since the data created by the "vector source" and "rational re-sampler" blocks are created all at once instead of at a set rate, a "throttle" block must be added to the output of the "rational re-sampler" block. The sampling rate on the "throttle" block is set to 8 kHz, which limits the rate at which data was sent to the rest of the flowgraph to 8 kHz. Without this block, the computer would attempt to use all available processing power to run the transmitter flowgraph until the computer froze. Furthermore, the sampling rate would be determined by the processing power of the computer.

In the MATLAB simulated transmitter, the *netshift* vector was then used as the phase argument of a cosine wave. In GNU Radio, this approach was less straightforward. Instead, a combination of multiplication and addition was utilized to recreate that same phase shifting effect. The output of the "throttle" block was connected to a "multiply by a constant" block. The constant used in the multiply block was the number two. For each sample created by the "throttle" block, this block multiplies the input by two. The output of the "multiply by a constant" block was then connected to the input of the "add a constant" block. The constant used in the addition block as the number negative one. The combined effect of these two blocks is to turn the phases indicated in the "vector source" (similar to those in the *netshift* vector) from bits into positive and negative ones.

After the "add a constant" block, the data stream was low pass filtered. The cutoff frequency of the filter was set at 15 Hz with a transition width of 5 Hz and the sampling rate was set at 8 kHz.

This low pass filter has the same role in the transmission of PSK-31 as the low pass filter used in the MATLAB transmitter – the filter is part of the PSK-31 standard and it prevents abrupt phase changes in the transmitter. Fig. 5.6 shows the PSK-31 signal after the data stream has been low pass filtered.

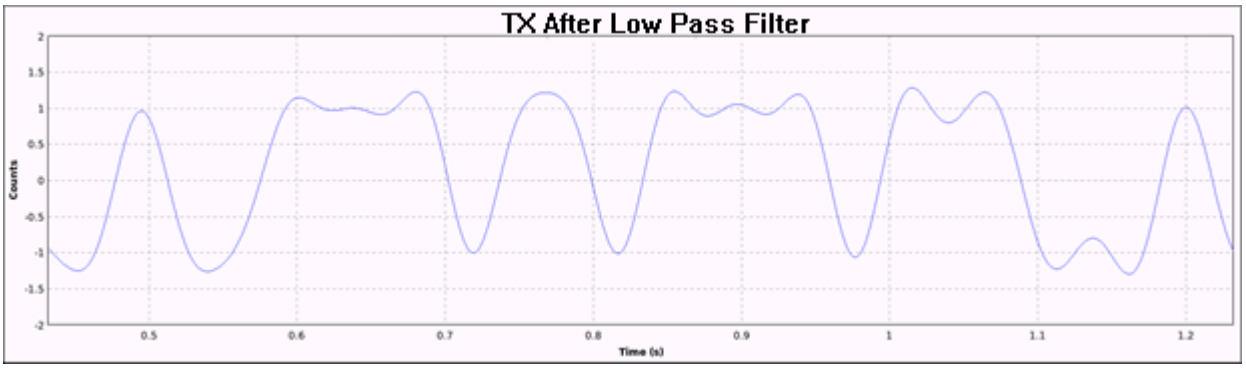

Figure 5.6 GNU Radio Scope Plot of PSK-31 after Low Pass Filtering for the Message "14"

The output of the low pass filter block was then inputted into a "multiply" block with the other input a cosine wave set using a variable slider with a default value of 1 kHz. While the flowgraph was running, the user could move the slider to adjust the subcarrier frequency of the slider. The slider proved that the subcarrier frequency could be set by a variable. Fig. 5.7 shows the PSK-31 signal after multiplication by the subcarrier.

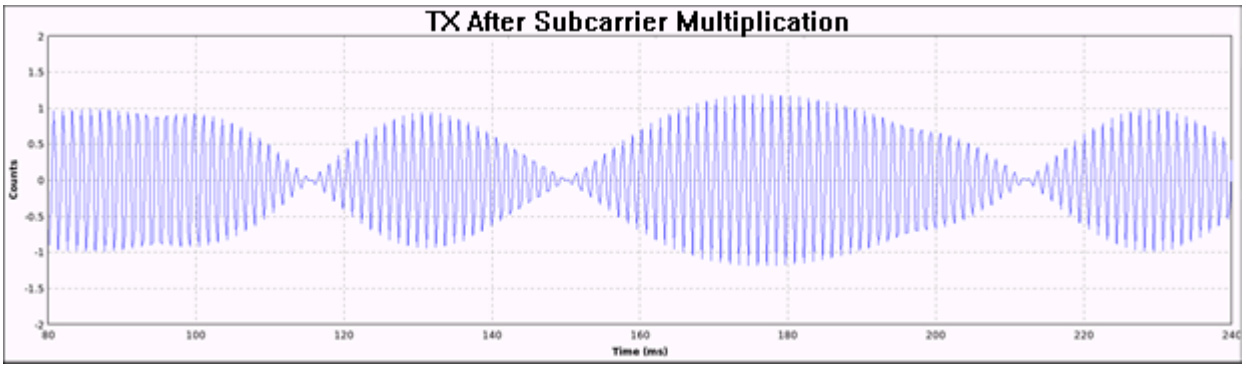

Figure 5.7 GNU Radio Scope Plot of PSK-31 after Multiplication by a Subcarrier for the Message "14"

Finally, the output of the "multiply" block was connected to an "audio sink" block set at a sampling frequency of 8 kHz. This block allowed the generated PSK-31 signal to play out of the audio output port on the USRP for testing purposes. When actually generating a PSK-31 signal, the output of the "multiply" block would be connected to a "USRP sink" that would upconvert the signal to radio frequency (RF).

Chapter 6: PSK31 Receiver

6.1 Simulation

Figure 6.1 Block Diagram for PSK31 Receiver

MATLAB was used to simulate a PSK-31 receiver due to familiarity with the software, ease of use and data manipulation tools. Once the receiver was verified to work in MATLAB, the code served as a template for recreating the receiver in GNU Radio. Code for the MATLAB receiver can be found in the appendix.

The first step in receiving a PSK-31 signal is determining the subcarrier frequency. For this simulation, MATLAB prompted the user to input the subcarrier frequency. Once the subcarrier frequency was known, the PSK-31 signal (simulated as a .WAV file) was read into the receiver as the vector *PSK31_TX*. The wavread function was used to read the file and the transpose function was used to create a single row-long vector that held all the samples in the .WAV file. Fig. 6.2 shows the MATLAB plot of the received .WAV file for the message "14".

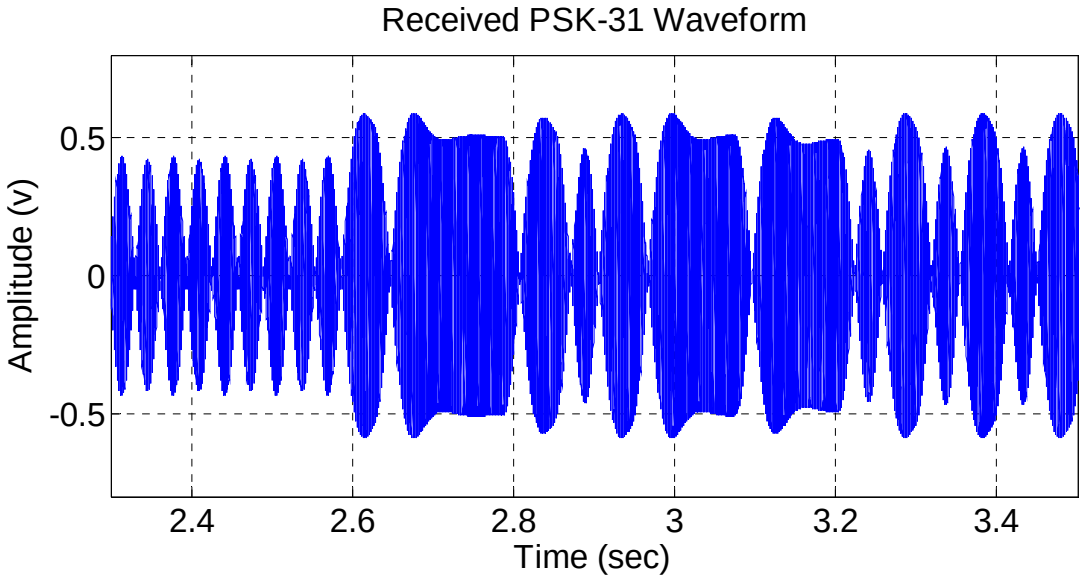

Figure 6.2 MATLAB Plot of the Received PSK-31 Signal for the Message "14"

Since the subcarrier frequency was set, the *PSK31_TX* vector was mixed down to complex baseband by multiplying the samples in the *PSK31_TX* vector by a cosine wave generated at the subcarrier frequency. To simplify the demodulation of a PSK-31 signal in MATLAB, phase synchronization was artificially forced. The sampling frequency used in all files was 16 kHz – the same sampling frequency used in the PSK-31 MATLAB transmitter simulations. Fig. 6.3 shows the PSK-31 signal after being mixed down to complex baseband.

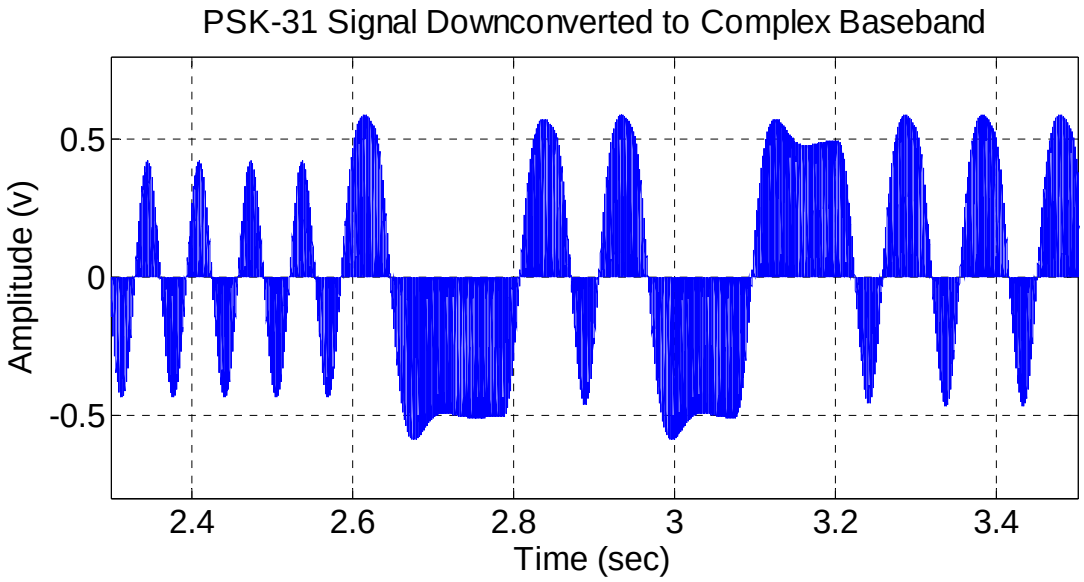

Figure 6.3 MATLAB Plot of the Received PSK-31 Signal Mixed Down to Complex Baseband for the Message "14"

The next step was to find the energy in each bit by integrating over a single bit period. Since the start of the file was known to be the start of a bit, no additional effort was needed to synchronize the receiver to the bit sequence in the *PSK31_TX* vector. Then, the energy under each bit was multiplied by the bit period (0.032 seconds) to find the energy in a single bit period. Fig. 6.4 shows the results of the integration and multiplication.

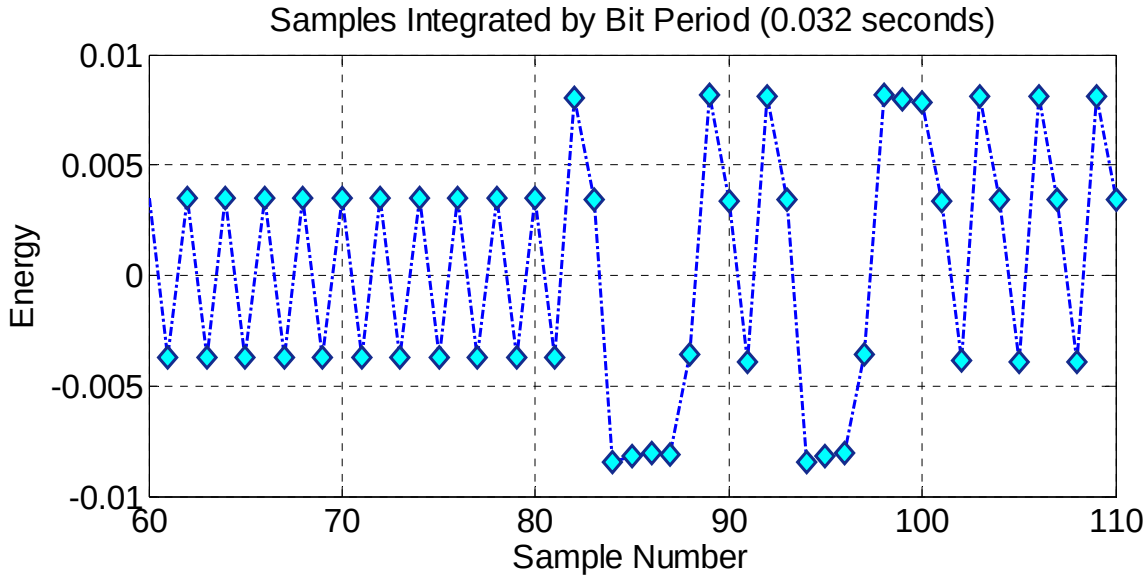

Figure 6.4 Plot of the Received Energy for each Bit Period in the Received Message "14"

The result of this operation was a series of positive and negative numbers. Using the MATLAB sign function, positive numbers were assigned the bit value '1' and negative numbers were assigned the bit value '0' and the result was stored in the vector *phase_value*. These assigned numbers reflect the phase of the PSK-31 signal and not the Varicode bits encoded within the signal because PSK-31 is differentially modulated.

To find the Varicode bits, the negation of the "exclusive or" operation was executed on the entire *phase_value* in groups of two. This means that if *phase_value* changed from positive to negative or from negative to positive, the Varicode bit being represented was a zero. Conversely, if the *phase_value* had two positive or two negative numbers in a row, then the Varicode bit being represented was a one. The results of this operation were stored in the vector *varicode_bits*. Table 6.1 shows the vectors *phase_value* and *varicode_bits* for the message "14".

phase_value	1	0	1	1	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	1	1	1	0
varicode_bits	0	0	1	0	1	1	1	1	0	1	0	0	1	0	1	1	1	0	1	1	1	0	0

Table 6.1 Vectors *phase_value* and *varicode_bits* for the Message "14"

Finally, a "for loop" was created to find the start and end of each Varicode character. Since each character must start and end with a '1' and at least two '0's separate characters, looking for the sequence '0 0' revealed the start and end of each bit sequence. Using this information, the Varicode sequences were extracted from the *varicode_bits* vector and converted into strings. This allowed a simple search function to be created that would look for the Varicode sequence using case statements and return the corresponding ASCII character. Once all the characters were found, the message was displayed. Fig. 6.5 shows the result that the MATLAB receiver output to the command window.

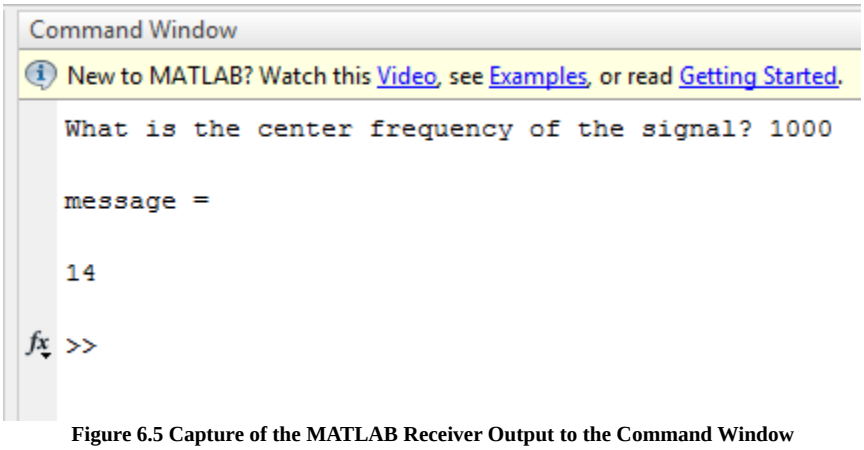

Figure 6.5 Capture of the MATLAB Receiver Output to the Command Window

6.2 Hardware Recreation

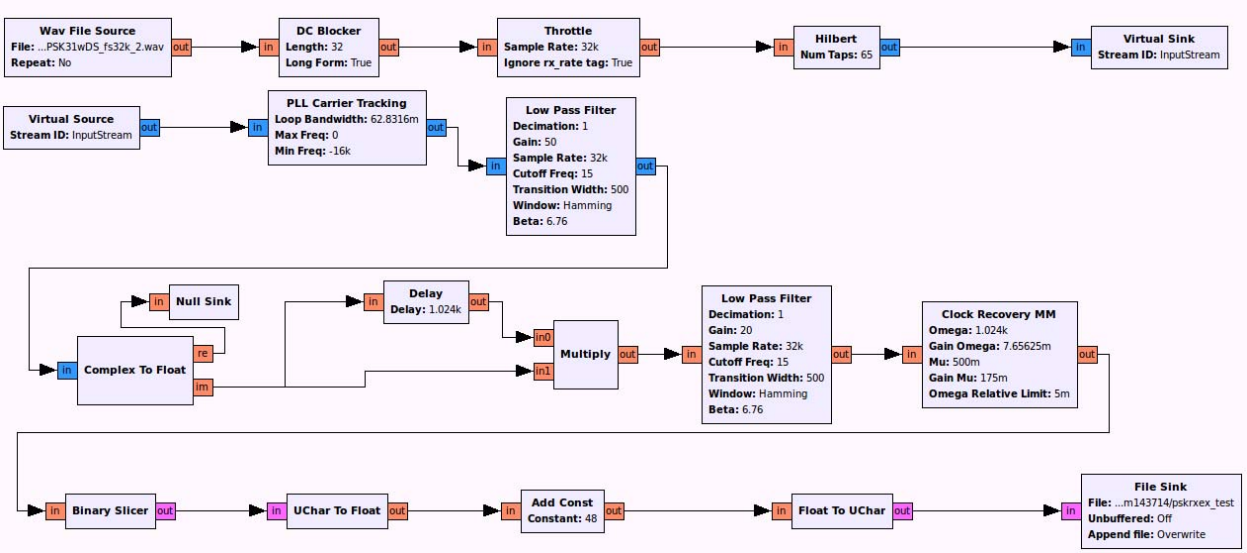

Figure 6.6 GNU Radio Companion Flowgraph for the PSK-31 Receiver

The PSK-31 receiver was constructed in GNU Radio using the code generated in MATLAB as a template. The complete GNU Radio flowgraph for the receiver is shown in Fig. 6.6. Instead of utilizing the USRP to capture PSK-31 transmissions over the air, the MATLAB version of the transmitter was used to create a .WAV file that contained a PSK-31 modulated signal with a message of, "The quick brown fox jumped over the lazy dog 0123456789". This approach allowed for controlled and repeated trials of the receiver. As a result, the first block in the receiver flowgraph was a .WAV source. After experimentation, it was determined that a DC offset in the wave file was causing problems with the receiver, so a DC blocker was added to the flowgraph. After these first two blocks, the stream of data was represented by floats. Fig. 6.7 shows the message "14" after the DC blocker.

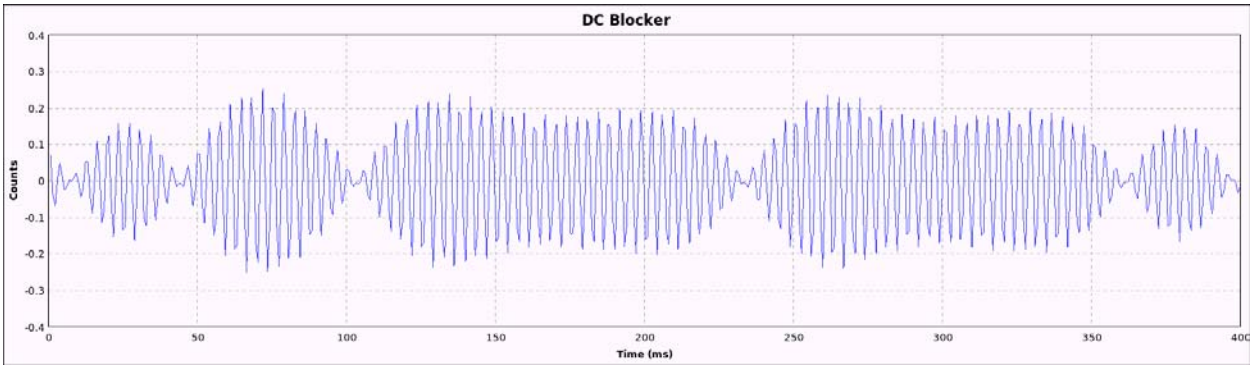

Figure 6.7 GNU Radio Scope Capture of PSK-31 Message "14" after DC Blocker

A phase-locked loop (PLL) was used to determine the subcarrier frequency. The PLL tracks the error between the incoming frequency and the output frequency to convert the signal to complex baseband. Figure 6.8 shows the block diagram for a PLL. An important feature of the PLL is that it requires a stream of complex numbers as its input. The Hilbert transform was utilized in order to convert the stream of floats into a stream of complex numbers. Figure 6.9 shows the

spectrum of the message “14” after the Hilbert transform (top) and after the PLL (bottom). Note that for an original subcarrier frequency of 1 kHz, the Hilbert transform creates an additional frequency peak at -1 kHz.

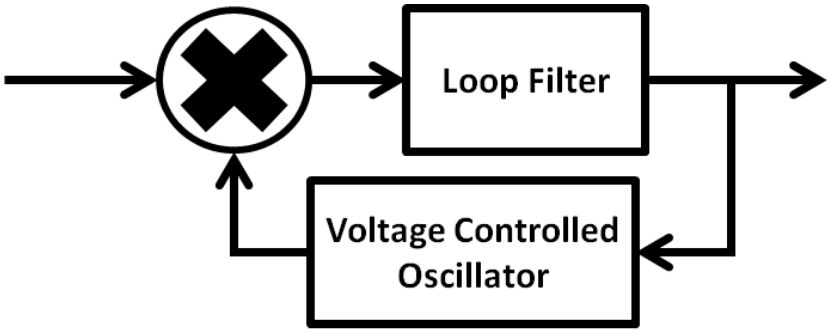

Figure 6.8 Block Diagram for a Phase-Locked Loop

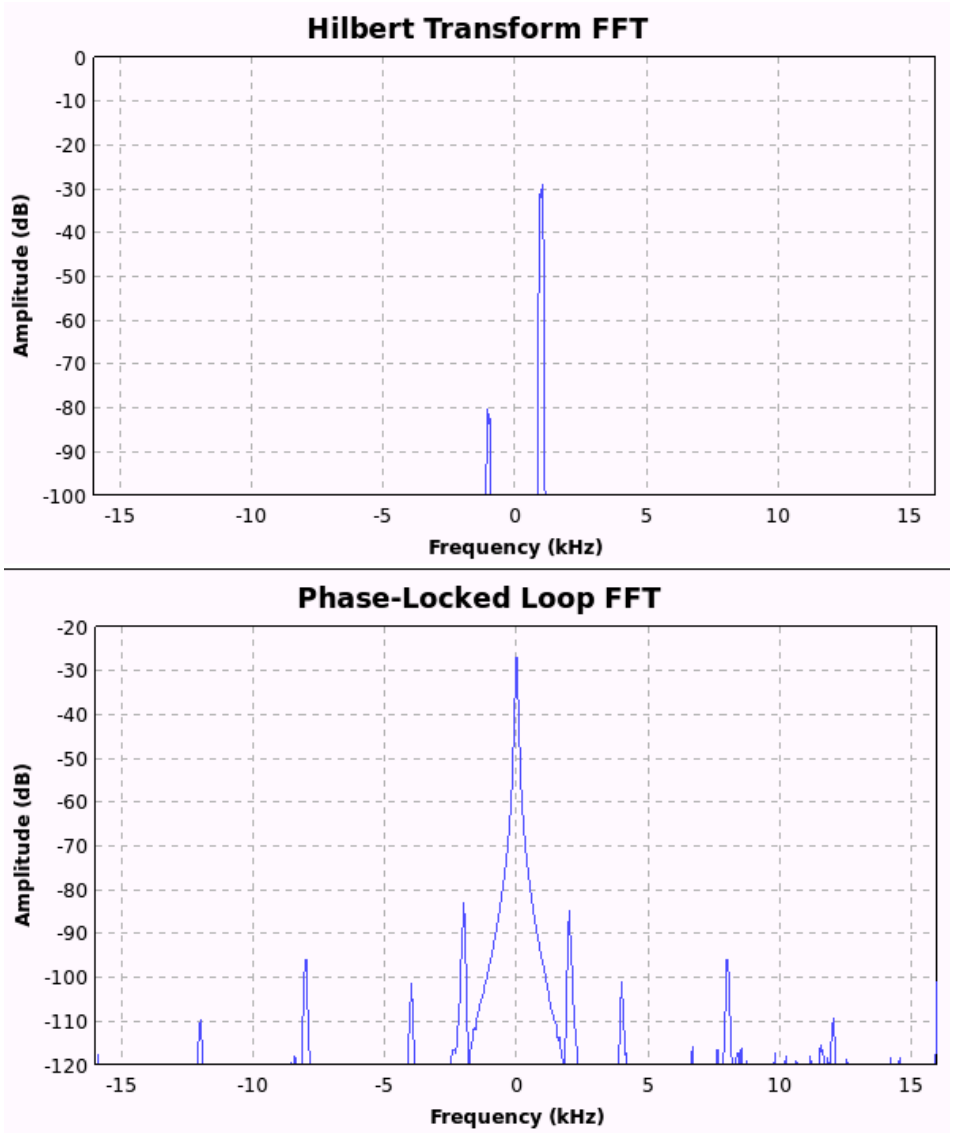

Figure 6.9 Frequency Spectra of the PSK-31 Message “14” after the Hilbert Transform (top) and the Phase-Locked Loop (bottom)

Eqn. 6.1 describes the Hilbert transform function mathematically. Conceptually, the Hilbert transform delays the entire signal by 90° - which was used to create a copy of the signal in the negative frequency domain [12]. In Fig. 6.9, the PSK-31 signal is seen to have a frequency peak at -1 kHz and 1 kHz. It is important to note that the Hilbert transform maintains the magnitude of the original signal.

$$x_h(t) = \frac{1}{\pi}\int_{-\infty}^{+\infty} \frac{x(\propto)}{t-\propto} d \propto \qquad \textbf{Equation 6.1}$$

Once the signal was at complex baseband, it was converted back into a stream of floats. Each item in the stream was then multiplied by a sample delayed by one bit period. This operation removed the differential encoding from the signal and left a stream of data where each bit was represented by multiple samples. Fig. 6.10 shows the message "14" after each item in the data stream was multiplied by a sample delayed by a single bit period.

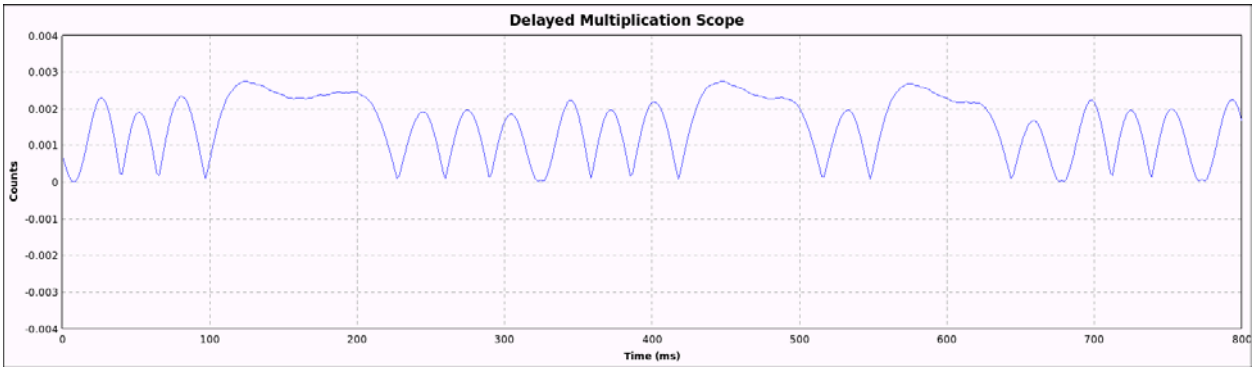

Figure 6.10 GNU Radio Scope Capture of PSK-31 Message "14" after Multiplication by a Sample Delayed by One Bit Period

The stream was then sent through a low-pass filter with a cutoff frequency of 15 Hz to eliminate any extraneous sidebands. Figure 6.11 shows the frequency spectrum of the signal after the phase-locked loop (top) and after it was sent through the low-pass filter (bottom). Note that the first set of frequency peaks around the baseband signal (centered around 0 Hz) have decreased in power from 50 dB less than the baseband signal to 100 dB less than the baseband signal.

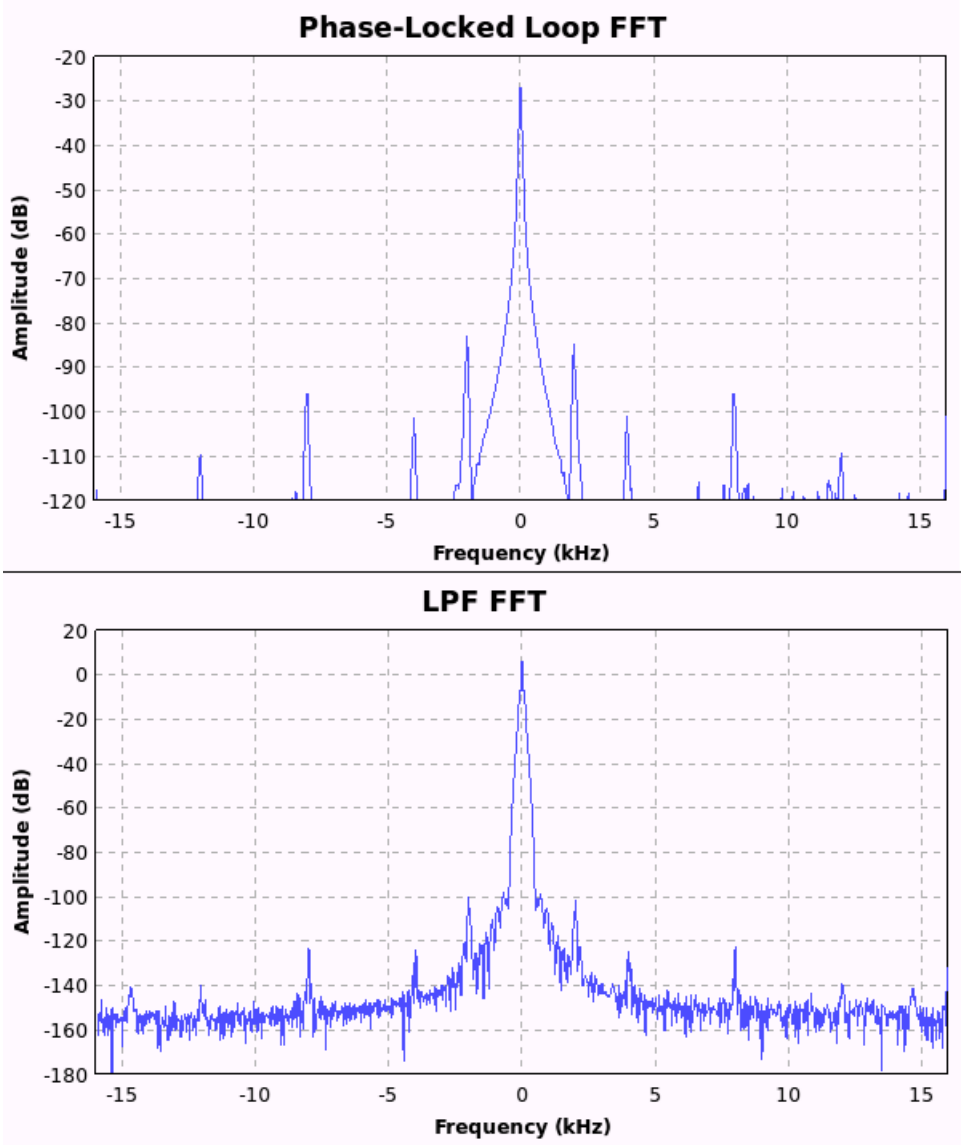

Figure 6.11 GNU Radio Frequency Spectrum of Message "14" after the Phase-Locked Loop (top) and after the Low Pass Filter (bottom)

The final operation in the receiver was to sample once during each bit period to extract the Varicode bits within the signal. For testing purposes, the bits were saved to a file as a stream of unsigned characters to allow for quick verification of the receiver's performance. Figure 6.12 shows the extra blocks required to convert the bits to ASCII characters readable in a .bin file.

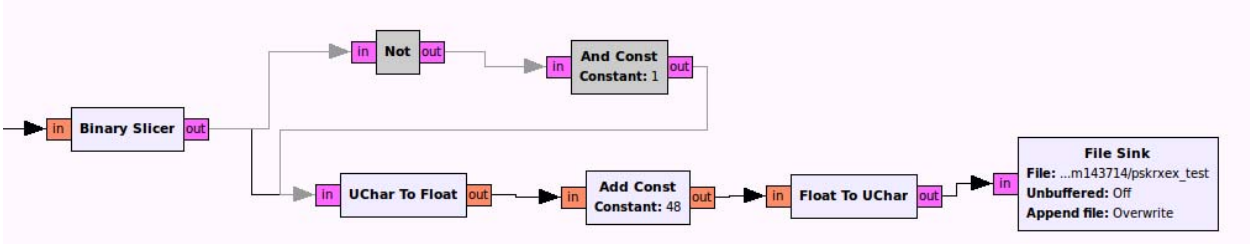

Figure 6.12 GNU Radio Flowgraph Chain for Conversion of Binary Data to .bin Readable Characters

Chapter 7: Testing and Results

7.1 Methodology

Testing of the project was broken into four distinct phases:

1. Transmitter testing with verification in Digipan
2. Receiver testing using .WAV file created from MATLAB transmitter
3. Doppler shift curve creation and estimation using International Space Station (I.S.S.) data as a reference
4. End-to-end system testing and verification

All simulation was initially done in MATLAB due to familiarity with the software. Once a step had been completed in MATLAB, it was repeated in GNU Radio and tested in a similar manner. Testing of the project followed a logical order that built each new step upon the previous steps. For example, once the transmitter design was finalized and verified in Digipan, .WAV files could be created using the transmitter to test the receiver.

7.2 Digipan

Digipan is commercially available software used by amateur radio operators to transmit and receive signals across a variety of standards, including PSK-31 [13]. Digipan uses the computer's soundcard to do some signal processing work and relies on input from a microphone between 1 kHz and 5 kHz. As long as the signal standard is supported, and in the range of frequencies that Digipan operates, it will be demodulated. Digipan is easy to use and a trusted tool of the amateur radio operator. Fig. 7.1 shows a Digipan capture of multiple PSK-31 transmissions.

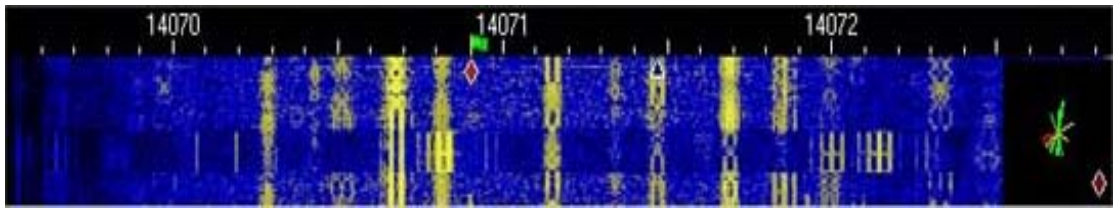

Figure 7.1 Digipan Capture of Multiple PSK-31 Messages

For this project, an additional advantage of Digipan was that it tolerates small levels of Doppler shift. To find what level of Doppler shift Digipan could handle, a fifteen second PSK-31 message was created ("the quick brown fox jumped over the lazy dog 1234567890") and altered to exhibit Doppler shift at a constant rate between 0 Hz/second and 6 Hz/second. Fig. 7.2 shows the received PSK-31 with a constant Doppler shift of 5 Hz/second. Table 7.1 shows the results of the test. Digipan was able to handle 2 Hz/second of Doppler shift before bit errors appeared in the received message.

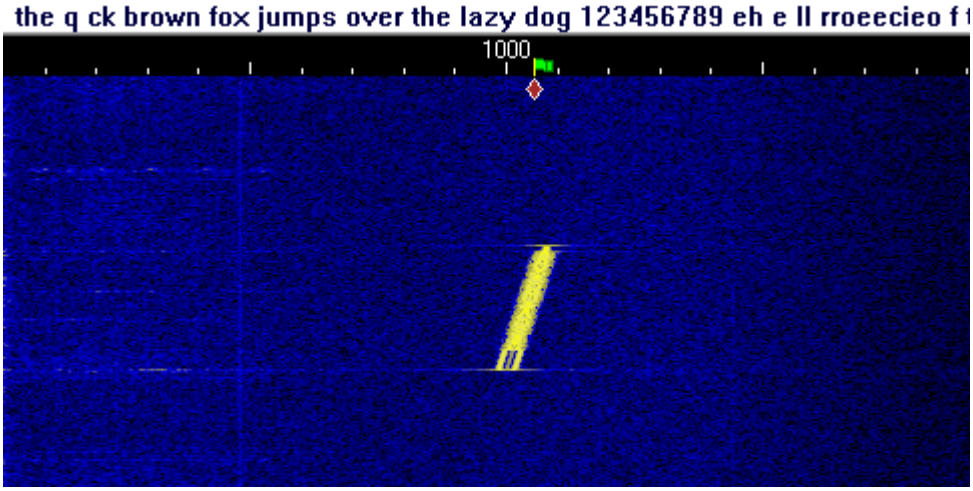

Figure 7.2 Digipan Capture of PSK-31 Signal with 5 Hz/second of Doppler shift

Doppler shift Rate	Result
0 Hz/sec	Test case; "the quick brown fox jumps over the lazy dog 123456789"
1 Hz/sec	Full message appears, just like test case
2 Hz/sec	Full message appears, just like test case
3 Hz/sec	About ½ of characters are errors; "thtrowo fox Bumps over th"
4 Hz/sec	About ¾ of character are errors
5 Hz/sec	Only 2 character errors; "the q ick brown fox jumps over the lazy dog 123456789"; Digipan requires the right tuning to get this result
6 Hz/sec	About ½ of characters are errors; received message is scattered across multiple channels

Table 7.1 Results for Digipan Doppler shift Tolerance Testing

7.3 Transmitter Testing

Initial testing of the transmitter came in designing the MATLAB simulation. Time and frequency plots of each step in the creation of the PSK-31 signal were used to streamline the design process. In order to test that the generated PSK-31 signal in MATLAB was valid, the signal was multiplied by a subcarrier frequency of 1 kHz and played through the computer's speakers. A second computer was set-up with a microphone to capture the PSK-31 signal in Digipan. When the message created in MATLAB matched the demodulated version shown in Digipan, the transmitter was completed. Verification gave the green light for implementing the simulation code from MATLAB and modifying it to work with GNU Radio. The GNU Radio transmitter was tested in a similar fashion. Multiple tests were run across a variety of subcarrier frequencies to ensure that the PSK-31 transmitters were operating as desired. Fig. 7.3 shows a successful test of the message "the goat is old and gnarly", transmitted at a subcarrier frequency of 1 kHz using the GNU Radio transmitter.

Figure 7.3 Digipan Capture of PSK-31 Signal Generated in GNU Radio

The over the air testing of the MATLAB transmitter proved beneficial to fully understanding the PSK-31 standard. Since no single reference could be found that described every facet of the standard, testing allowed pieces of PSK-31 to be put together until Digipan validated the signal. For example, an early reference acknowledged that PSK-31 needed to filter the baseband bits before subcarrier multiplication, but did not specify how. Various windowing functions and filters were tested before it was discovered that PSK-31 required a low pass filter with a cutoff frequency of 15 Hz.

7.4 Receiver Testing

Since the signals created by the transmitters in MATLAB and GNU Radio were verified to be accurate PSK-31 transmissions, these signals could be used to test the receiver design. PSK-31 signals created in MATLAB were saved as .WAV files that the receiver would read. Both the MATLAB and GNU Radio receivers could be tested using these .WAV files.

In designing the MATLAB receiver, some major simplifying assumptions were made: the receiver knew what the subcarrier frequency would be and that the beginning of the .WAV file would exactly mark the start of the first bit period. As a result of these assumptions, the MATLAB receiver and GNU Radio receiver differ significantly.

Firstly, the GNU Radio receiver required the use of a phase-locked loop to determine the subcarrier frequency of the incoming PSK-31 signal. Two blocks were available in GNU Radio that seemed suitable for the task: a Costas loop block and a phase-locked loop carrier tracking block. Each block was tested using to determine how well it could track changes in frequency. Fig. 7.4 shows the flowgraph utilized for testing the phase-locked loop.

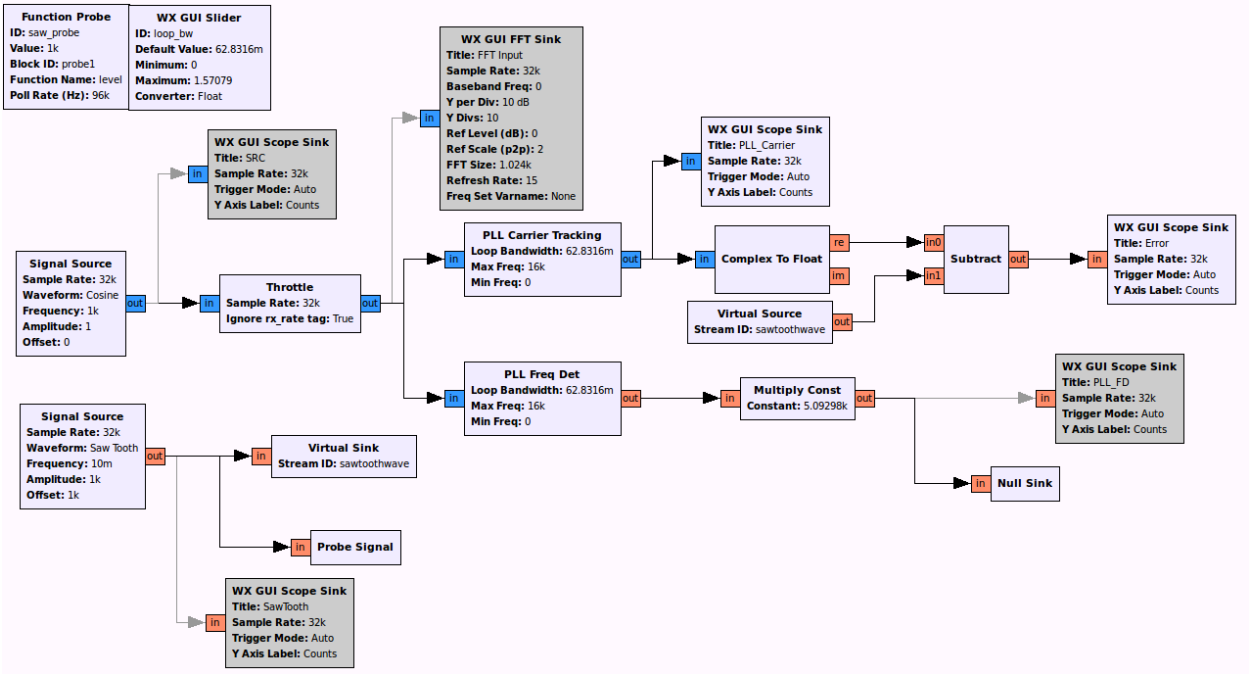

Figure 7.4 GNU Radio Flowgraph Used for Testing the Costas Loop and Phase-locked Loop Carrier Tracking Blocks

For the test, the loop bandwidth of the Costas loop and phase-locked loop carrier tracking block was set at 2*pi/100. Although the Costas loop was able to respond to frequency changes more rapidly and with greater resolution than the phase-locked loop carrier tracking block, the Costas loop had an effective range that depended on the sampling rate. For a given sampling rate, f_s, the effective range of the Costas loop, f_{range}, was given by Eqn. 7.1.

$$\frac{f_s}{2*\pi} < f_{range} < \frac{f_s}{100} \qquad \textbf{Equation 7.1}$$

For a Doppler shift application where the subcarrier frequency would pass from positive frequencies, through DC to negative frequencies, the Costas loop could not be used. Once this decision had been made, the loop bandwidth of the phase-locked loop was altered to 10*pi/100 so that the receiver would be able to track frequencies over the expected range of Doppler shift (+/- 12 kHz).

As part of using a phase-locked loop in GNU Radio, the .WAV file source had to be converted to a stream of complex numbers by using the Hilbert transform function. Simply using a "float to complex block" and ignoring the imaginary argument would result in the phase-locked loop outputting the same signal that it was given as an input instead of converting the signal to complex baseband.

The second major change to the GNU Radio receiver was altering the block diagram used to demodulate PSK-31 to account for functionality available in GNU Radio. In the MATLAB receiver, the complex baseband signal was integrated over each bit period, evaluated to determine the bits and then the negation of the exclusive or operation was taken to leave the Varicode bit sequence as a result. In the GNU Radio receiver, this approach was altered to remove the heavy processing requirement of integration. Instead, the first step was to multiply each of the samples

in the complex baseband signal by a sample delayed by one bit period. The signal was then sent through a low pass filter and a clock synchronization block was used to pick the Varicode bits out of the samples.

While the MATLAB receiver was only tested using .WAV files generated by the transmitter, the GNU Radio receiver's final test used a "live" transmission of PSK-31 from the MATLAB transmitter. The headphone jack of the computer playing the PSK-31 signal output by the MATLAB transmitter was connected to the "line in" port of the USRP E100 using a standard audio cable. Successful completion of this test proved that the GNU Radio receiver could perform clock synchronization even when the start of the first bit period was unknown. The GNU Radio receiver was able to successfully decode the message "the quick brown fox jumped over the lazy dog 0123456789" without any bit errors. Fig 7.5 shows a plot of the received signal at complex baseband and the resulting bits.

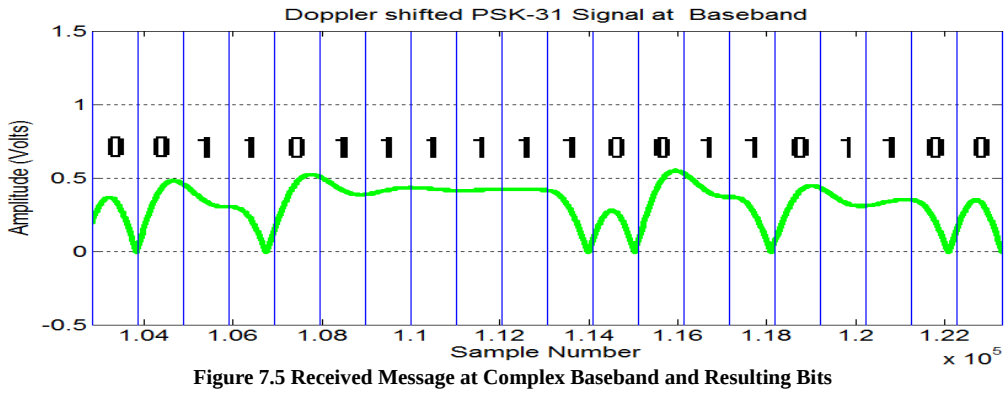

Figure 7.5 Received Message at Complex Baseband and Resulting Bits

7.5 Doppler Shift Curve Creation and Estimation

Initial Doppler shift data for low-Earth orbit (LEO) was gathered in the USNA Satellite laboratory from transmissions of the International Space Station. These curves were used to verify that the Doppler shift curves created in MATLAB appeared to match the range and extent of true Doppler shift curves.

Doppler shift curves were then generated for the intended satellite orbit for an expected event horizon of 3000 km. 100 curves were generated with samples taken at a rate of 16 samples per second. MATLAB was used to generate an m-ary curve estimator that could take a random piece of a random Doppler shift curve and find which curve that piece was taken from and when on that curve the piece began. Using this information, the curve estimator could return the value of the next projected value of Doppler shift that the satellite was expected to generate. The M-ary curve estimator was coded as a custom block in GNU Radio and print statements were used to evaluate its performance. Fig. 7.6 shows a screen capture of some of the print statements created by the M-ary curve estimator. For the test shown in Fig. 7.6, 5 samples are being recorded to estimate future values of Doppler shift and the curve estimator returns the best estimate for the next value of Doppler shift that the satellite will exhibit.

```
*****************************************
sample at b-4 location: 10999.7
sample at b-3 location: 11005.4
sample at b-2 location: 11002.5
sample at b-1 location: 11001.4
sample at b-0 location: 10998.2
noutput_items: 16
*****************************************
Guessing that the next sample is
curve number: 1
segment number: 1561
Data_ptr at segment_guess: 10998.1

*****************************************
sample at b-4 location: 11005.4
sample at b-3 location: 11002.5
sample at b-2 location: 11001.4
sample at b-1 location: 10998.2
sample at b-0 location: 10996.1
noutput_items: 16
*****************************************
Guessing that the next sample is
curve number: 3
segment number: 1574
Data_ptr at segment_guess: 10996.1

*****************************************
sample at b-4 location: 11002.5
sample at b-3 location: 11001.4
sample at b-2 location: 10998.2
sample at b-1 location: 10996.1
sample at b-0 location: 10995.5
noutput_items: 16
*****************************************
Guessing that the next sample is
curve number: 1
segment number: 1595
Data_ptr at segment_guess: 10995.4
```

Figure 7.6 Terminal Window Capture of Output from M-ary Curve Estimator

Various levels of noise were then added to the system and the performance of the M-ary curve estimator was measured over 1000 tests for various lengths of randomly generated samples drawn from the Doppler shift curves. A successful test was only when the curve estimator could exactly pinpoint the curve and starting segment for the sample. The results were calculated as a percentage of the 1000 tests that the curve estimator was exactly accurate and plotted against the length of the sample for each noise level. Fig. 7.7 shows the MATLAB plot of the results.

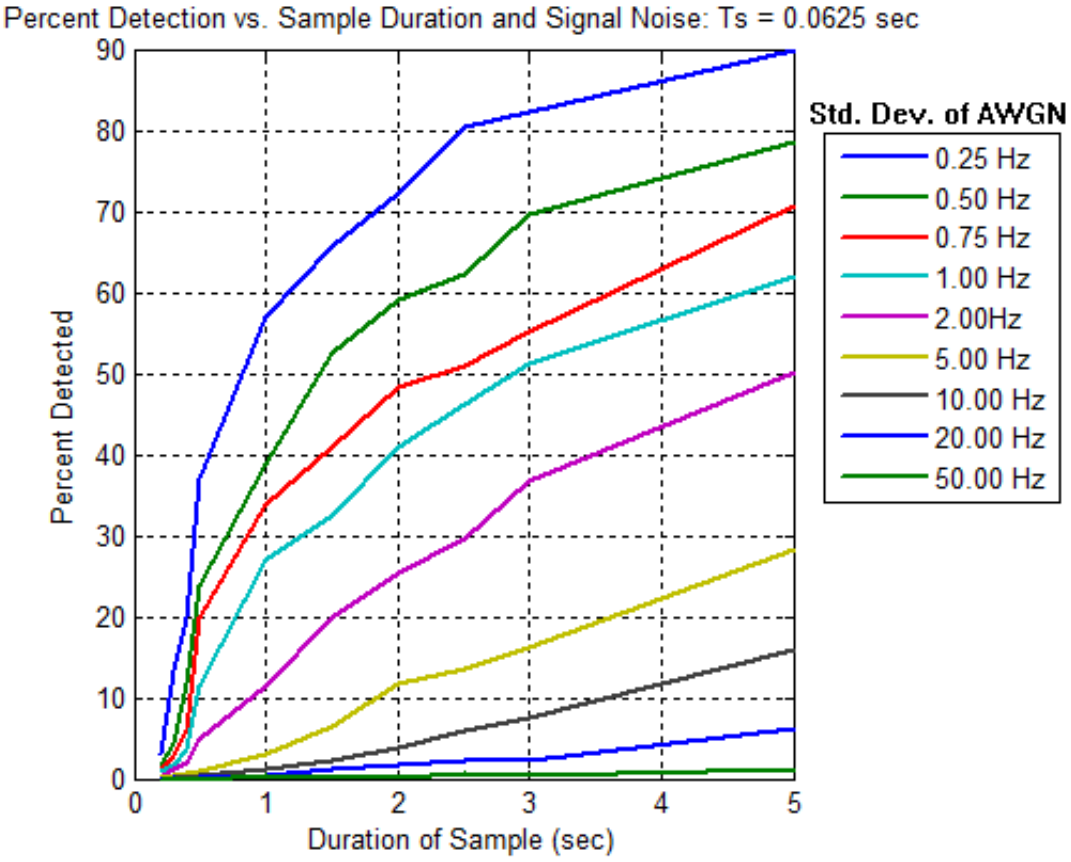

Figure 7.7 MATLAB Plot of the M-ary Curve Estimator's Ability to Detect Doppler Shift in Noise

Since Digipan can tolerate Doppler shift at levels up to 2 Hz/second, the probability of detection curves were regenerated. Eqn. 2 calculates what the tolerance level of Digipan given the duration of the sample used to estimate Doppler shift (from 0.125 seconds to 5 seconds) and the constant 2 Hz/second of Doppler shift that Digipan could handle.

$$Tolerance = (Sample\ Duration\ [sec]) * 2\ [\frac{Hz}{sec}] \qquad \textbf{Equation 7.2}$$

A successful test occurred whenever the difference in frequency between the start segment estimation given by the curve estimator and the actual starting segment was less than the value given in equation 2. Figure 7.8 shows the MATLAB plot of the probability of successful detection by the m-ary curve estimator given the tolerance of Digipan. Note that while the lowest rate of detection in Fig.7.7 occurs for additive white Gaussian noise (AWGN) with a standard deviation of 50 Hz, the lowest rate of detection for Fig. 7.8 occurs for AWGN with a standard deviation of 1 kHz.

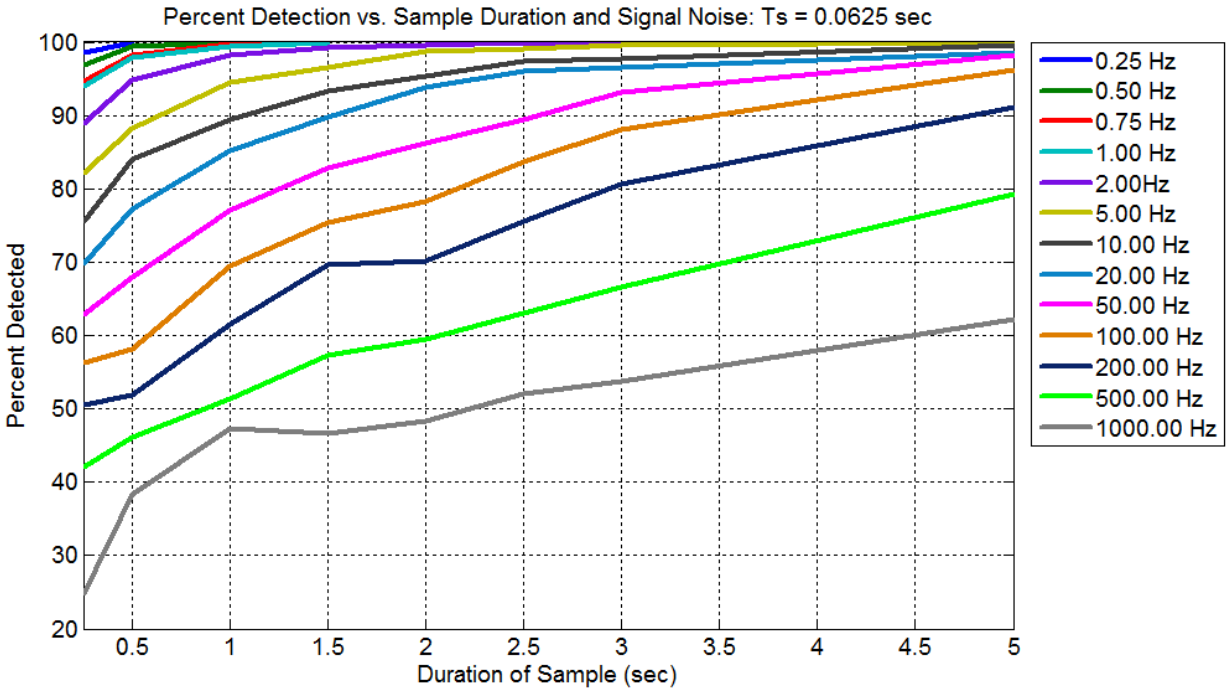

Figure 7.8 MATLAB Plot of the M-ary Curve Estimator's Ability to Detect Doppler Shift in Noise given Digipan's Tolerance

7.6 End-to-end System Testing and Verification

Complete testing of the system involved connecting the receiver to the transmitter and adding in the curve estimator. While the actual system would use the results of the curve estimation to determine the subcarrier frequency utilized by the transmitter, the test set the subcarrier frequency of the transmitter to a constant 1 kHz so that Digipan could test the entirety of the system. Separate verification of the output produced by the curve estimator was completed to ensure that the whole system could as a single unit.

The majority of the end-to-end testing, however, used Doppler shift altered PSK-31 signals to test the performance of the receiver. While the receiver was able to bring the PSK-31 signals down to complex baseband, the clock synchronization block failed when Doppler shift was introduced into the PSK-31 signals. To measure the accuracy of the receiver, the result of the low pass filter that preceded clock synchronization in the GNU Radio receiver was plotted against time and printed out. Then, the bits were manually decoded and compared to the bits that should have been produced. The Doppler shift from the direct overhead pass was used to generate two test cases as seen in Fig. 7.9. Table 7.2 lists the characteristics of each test case.

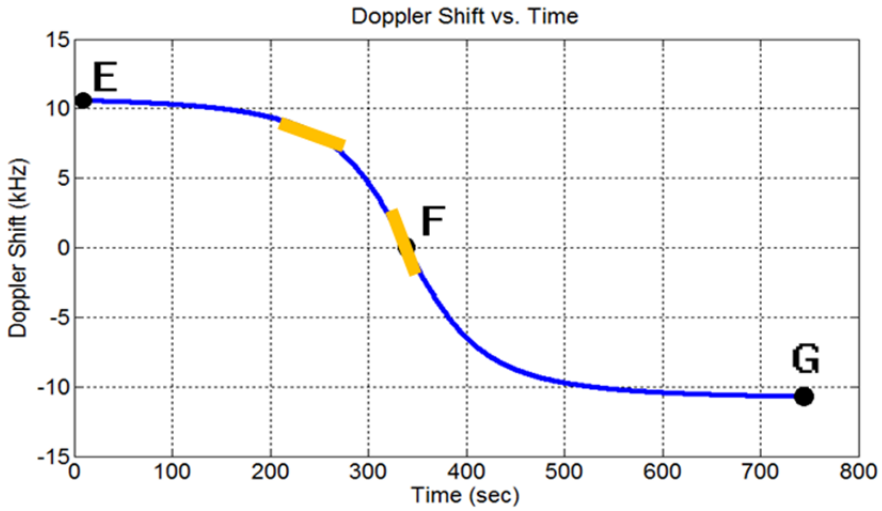

Figure 7.9 Side-by-side Comparison of Manually Decoded Message and the Expected Message

Test Case Description	Initial Doppler shift Frequency	Final Doppler shift Frequency	Change in Doppler shift Frequency	Rate of Change in Doppler shift
Relatively flat piece of curve in yellow on left of Fig. 7.9	10203.70 Hz	9562.03 Hz	641.67 Hz	42.78 Hz/sec
Relatively steep piece of curve in yellow on right of Fig. 7.9	4432.44 Hz	1447.87 Hz	2984.57 Hz	198.97 Hz/sec

Table 7.2 Numerical Characterization of Doppler shift used in PSK-31 Receiver Test Cases

For both test cases, manual decoding of the PSK-31 signal at complex baseband resulted in zero bit errors. Although complete automation of the process is currently limited by the receiver's ability to decode the bits in the PSK-31 signal at complex baseband, the two tests demonstrated that the satellite could receive signals and correctly demodulate signals affected by Doppler shift.

Chapter 8: Future Work

8.1 Stand-Alone PSK-31 Transmitter

One improvement currently in the works for the transmitter is the ability to take user input from a GUI and use that as the message the transmitter will send. This feature is called asynchronous input because the user will not always type at a constant rate. The user will start and stop typing at certain points in his message, which creates some problems in GNU Radio. GNU Radio blocks output data to a buffer that consecutive blocks read as an input buffer. When these buffers overfill or run dry, both problems that can occur with asynchronous input, GNU Radio does not behave in a desirable manner. With the upgrade of GNU Radio to patch 3.7, asynchronous systems are now feasible by using stream tags.

In order to bring asynchronous input into GNU Radio, a custom block must be made. Creating a block in GNU Radio is an extensive process that requires knowledge of out-of-tree modules. An out-of-tree module is a directory where test code and blocks can be developed separate from the core functionality of GNU Radio. Developing custom blocks requires a strict process to be followed and complete understanding of how a block must be formatted in C++ or Python. Furthermore, the block must be complete with an xml description if it is going to be utilized in the GNU Radio Companion.

Going one step further, adding a GUI to the keyboard source block would allow the user to hotkey certain presses that are common in the amateur radio community, such as inputting a call-sign or calling up another station. A fully functional GUI could even allow the user to alter the sub-carrier frequency of their transmission in real-time.

8.2 Utilize Stream Tags to Insert the "Tail"

Currently, the PSK-31 transmitter does not add the 750 milliseconds of unmodulated carrier to the end of the transmitted signal. This feature is not completely necessary, because the receiver is only looking for bits in the transmission and there are no bits in the tail. The tail does help to prevent the last couple characters from being warped at the receiving end, but this issue is currently solved by padding the end of the message with additional space characters.

In order to add the "tail" on the end of the PSK-31 transmission, stream tags need to be utilized. Stream tags are a relatively new feature in GNU Radio that I have yet to fully explore. A condition would be set that would look for the end of text (ETX) character. When this condition was met, a gate would switch to transmit the unmodulated subcarrier frequency of 0.75 seconds. Another condition would then need to be set to stop the tail from transmitting after the 0.75 seconds concluded.

At this point in time, the "tail" feature was not deemed critical in the operation of this project.

8.3 Multi-Channel Capability

Currently, the system is designed for use by a signal operator at any given time. This greatly

reduces the ability of any two users to communicate with each other during a single overhead pass of the satellite. With less than a ten minute window to communicate with the satellite during each pass, users must be able to communicate simultaneously.

In order to accommodate multiple users, some aspects of the project would have to be redesigned. The simplest foreseeable way to accomplish this goal would be to designate certain frequencies that could be used and run another instance of the project over each band. This approach is, however, computation intensive and unlikely to work for the large number of users that could theoretically be supported by such a narrowband signal.

8.4 Receiver Automation

Currently, the receiver is unable to perform the clock recovery operation necessary to sample the bits at the correct intervals. While this step is critical for end-to-end performance of the communications system, it is a single step that can be accomplished at a later date. Verification of the receiver design requires that the bits be manually sampled by the Trident Scholar. In no way does the unavailability of this functional block within GNU Radio indicate that the project is not possible to complete. It is merely a reflection of the current lack of a block within GNU Radio that can perform the desired function.

References

[1] *History of Amateur Radio*. Rep. Alaska Hamfest. Web. 2 Dec. 2013. <http://www.rcarc.org/presentations/alaska/history.pdf>.

[2] Krebs, Gunter D. "OSCAR 1, 2." *Gunter's Space Page*. N.p., 30 Mar. 2014. Web. 06 Apr. 2014.

[3] Martinez, Peter. PSK31: A New Radio-teletype Mode with a Traditional Philosophy. N.p., n.d. Web. 12 Apr. 2014. <http://det.bi.ehu.es/~jtpjatae/pdf/p31g3plx.pdf>.

[4] Martinez, Peter. PSK31: A New Radio-teletype Mode with a Traditional Philosophy.

[5] Martinez, Peter. PSK31: A New Radio-teletype Mode with a Traditional Philosophy.

[6] Turner, Clint. The "CT" MedFer Beacon. N.p., 3 Nov. 2006. Web. 12 Apr. 2014. <http://www.ka7oei.com/psk_xmit_source2.html>.

[7] Neader, Scott. "PSK31 FREQUENCIES." PSK31 FREQUENCIES. N.p., n.d. Web. 22 Oct. 2013. <http://www.qsl.net/darn/PSK31.htm>.

[8] Martinez, Peter. PSK31: A New Radio-teletype Mode with a Traditional Philosophy.

[9] "ARRL." Amateur Radio on the International Space Station. ARRL, n.d. Web. 30 Sept. 2013. <http://www.arrl.org/amateur-radio-on-the-international-space-station>.

[10] Peat, Chris. ISS - Orbit. Heavens Above, n.d. Web. 12 Apr. 2014. <http://www.heavens-above.com/orbit.aspx?satid=25544>.

[11] Volcano Chimborazo. Volcano Discovery, n.d. Web. 12 Apr. 2014. <http://www.volcanodiscovery.com/chimborazo.html>.

[12] Langton, Charan. "Hilbert Transform, Analytic Signal and the Complex Envelope." SIGNAL PROCESSING & SIMULATION NEWSLETTER (1999): n. pag. Illinois Institute of Technology College of Engineering. Web. 12 Feb. 2014. <http://www.ece.iit.edu/~biitcomm/research/references/Other/Tutorials%20in%20Communications%20Engineering/Tutorial%207%20-%20Hilbert%20Transform%20and%20the%20Complex%20Envelope.pdf>.

[13] Teller, Skip. Digital Panoramic Tuning with a Windows PC. N.p., 4 Jan. 2000. Web. 12 Apr. 2014. <http://www.digipan.net/>.

Appendices

Appendix A: GNU Radio Tutorial

GNU Radio Installation and User Guide

What is GNU Radio?
GNU Radio is an open-source platform for the communications field. Imagine taking all the signal processing blocks from Simulink and all the communications functions from MATLAB and putting them into a free package and you start to scratch the surface of what GNU Radio actual contains. There is a graphical user interface (GUI) called GNU Radio Companion where users can drag and drop pre-made blocks to create communications systems. Users can even define their own blocks using Python code. If a user wants to dive deeper into their project, they can work entirely in Python or even in C++.

One of the major distinctions of GNU Radio is that, as open-source software, anything created in GNU Radio can be made to work with any piece of hardware. This being said, some hardware is more compatible with GNU Radio. The Universal Software Radio Peripheral (USRP) series, developed by Matt Ettus, is designed specifically for use with GNU Radio. Additional software, called the Universal Hardware Driver (UHD) links the USRP with anything created in GNU Radio. A variety of USRP devices exist for different purposes. This project implements the E100 with the basic transmitter and receiver daughterboards.

Additionally, the GNU Radio community is a large, evolving body. Tutorials, examples and function blocks are all available through the forum pages, YouTube and across the internet. The community is available to answer questions and help troubleshoot problems.

What is all of this actually used to accomplish?
Before that question is answered, think about cell phones in the late ‘80s or early ‘90s. They could only do one thing – call people. If you wanted to send messages to somebody, you would have to buy a pager and then you would have to lug around two devices instead of one. Today’s phones, however, can do both of these things and more! They can be used to call, text, surf the web and play video games – all on the same device. The way the phone works depends on which application – or program – is in use.

This is called software-defined radio (SDR), which is exactly the purpose of GNU Radio and the USRP. The USRP is the hardware whose function is defined by the user through GNU Radio. The UHD is simply used to translate the user’s instructions from GNU Radio to something that the USRP can understand.

How do I get started?
Before getting started, make sure that you have at least 500 MB worth of free space on your hard disk and a fair amount of time. On a typical computer set-up, it takes about two hours to install UHD and GNU Radio.

Installing GNU Radio with UHD can be done easily on any Ubuntu or Fedora machine by

opening a terminal window and running the following command from the GNU Radio website:

```
wget http://www.sbrac.org/files/build-gnuradio && chmod a+x ./build-gnuradio && ./build-gnuradio
```

This will remove any previous installations of UHD or GNU Radio, download the current stable version and then build the software from source code. If done correctly, the terminal window should spit out text similar to fig. 1.

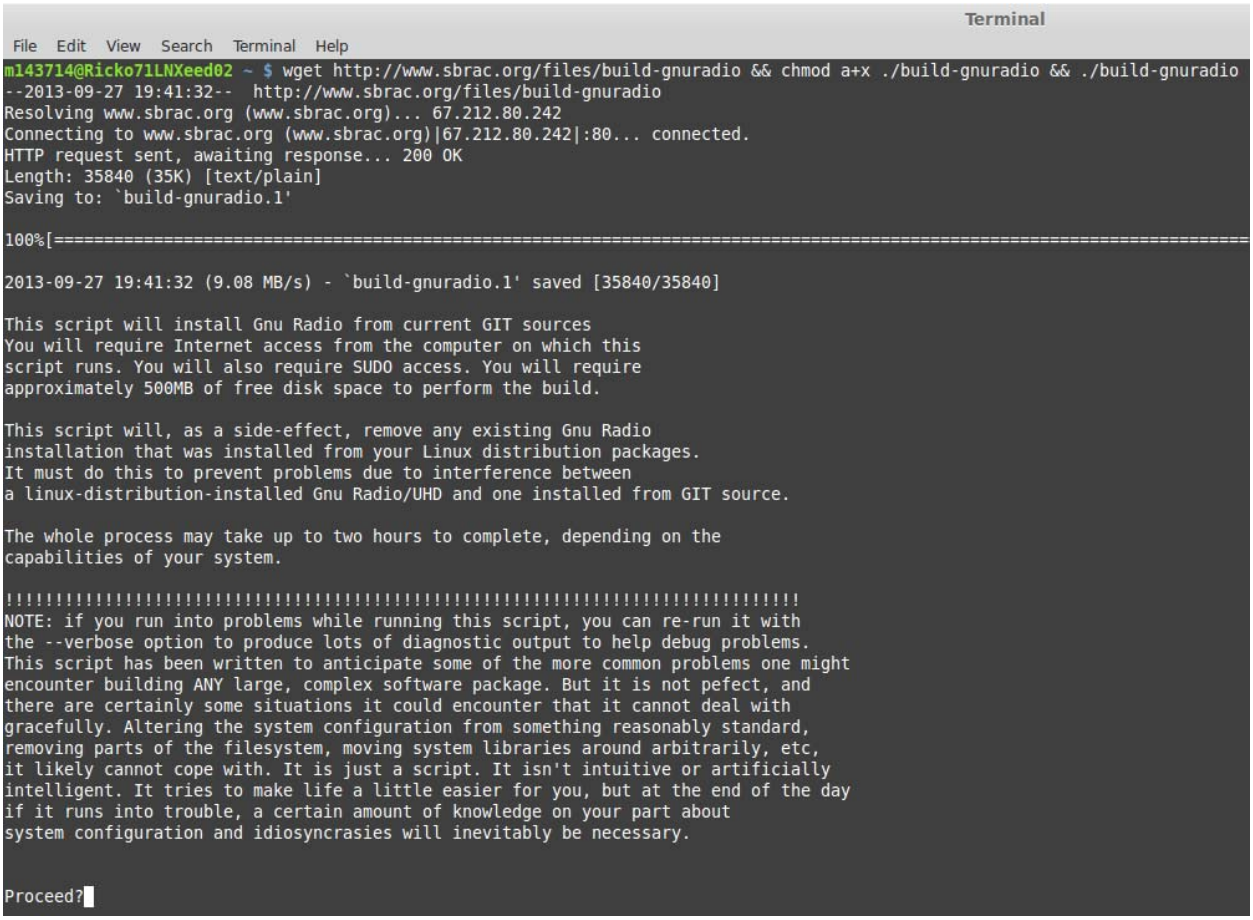

Fig. 1 Build GNU Radio Script

The build radio script takes a long time to run. After a successful build, the terminal window will output some text similar to fig. 2. Also, the script will tell you how to set the PYTHONPATH. This is the pathway that the computer will use to look for Python. Execute the command shown and GNU Radio will be ready to operate!

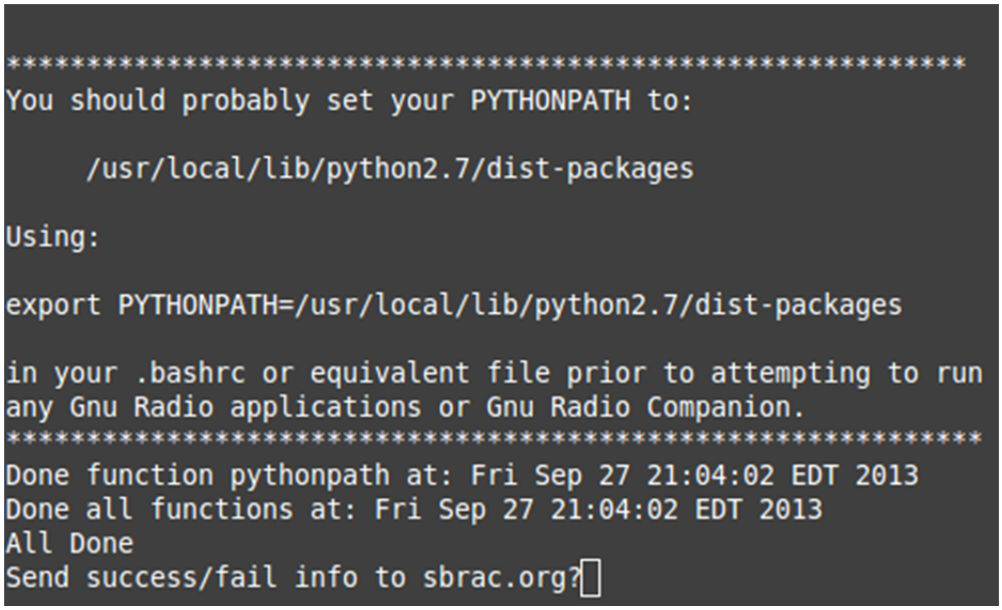

Fig. 2 Successful Build of GNU Radio

If your system is not running Fedora or Ubuntu (i.e. Windows) or you would prefer to manually build GNU Radio from source or even install an older binary version, the GNU Radio installation guide has a set of instructions that will help you in your endeavor. An important item to note is that you must install UHD from Ettus Research before you build GNU Radio from source if you want to use GNU Radio with your favorite USRP device.

Now that I have GNU Radio installed, how do I connect to my favorite USRP device?
First of all, it depends what sort of USRP device you have. Any USRP device that is in the "E" series is designed to operate as its own computer – the "E" stands for embedded, as in embedded processor. All other USRP devices must be controlled by a computer through an Ethernet connection. This section will cover how to establish communications with a USRP2 device and the same concepts should apply to all other USRP devices that are not in the "E" series.
Once the USRP is connected, the command "find_uhd_devices" should be able to locate the USRP. If this is unsuccessful, you can specify the IP address of the USRP be using the command:

```
find_uhd_devices --args="addr=192.168.10.2"
```

If this does not prove to be successful, then the IP address may be something other than 192.168.10.2. The first step in setting the desktop IP address using:

```
sudo ifconfig eth0 192.168.10.1
```

Next, the following command must be executed from the /uhd/host/utils directory:

```
sudo ./usrp2_recovery.py --ifc=eth0 --new-ip=192.168.10.3
```

This sets the IP address of the USRP2 to 192.168.10.3. If this is not successful, then the FGPA image on the USRP may need to be updated. Ettus Research has guides and forums that cover how to update FPGA files on each of their devices.

Once communication has been established with the USRP, you can create a simple FM receiver by following the YouTube video at http://www.youtube.com/watch?v=KWeY2yqwVA0. Note that this implementation will not work on the E100 or E110.

Well, what would I have to do to communicate with my new USRP E100?
The USRP "E" series has an embedded processor built into the device that allows the E100 and E110 to be used as its own computer. This means that unlike previous USRPs, the "E" series does not require a separate computer to control the USRP. Furthermore, this changes the way that the USRP is used. Software can be developed in GNU Radio on the USRP, or it can be developed on another machine and copied onto the USRP.

To set-up the E100 as a stand-alone computer, you will need a monitor that can operate from a DVI port and a keyboard. If you also want to use a mouse, then you will need a USB splitter. The E100 ships with an ADP-to-USB cable that will also be necessary. Plug the monitor into the DVI port, and the ADP-to-USB cable into the "USB host" port. From here, the USB splitter can be connected or you can plug in the keyboard directly (if you do not need to use a mouse). Power the device and you can get off running. The default username is "root" and the default password is "usrpe". UHD and GNU Radio come preinstalled on the E100. Fig. 3 shows the E100 set-up with a mouse and keyboard attached via a four way USB splitter. Fig. 4 zooms in on the E100 and the ports utilized.

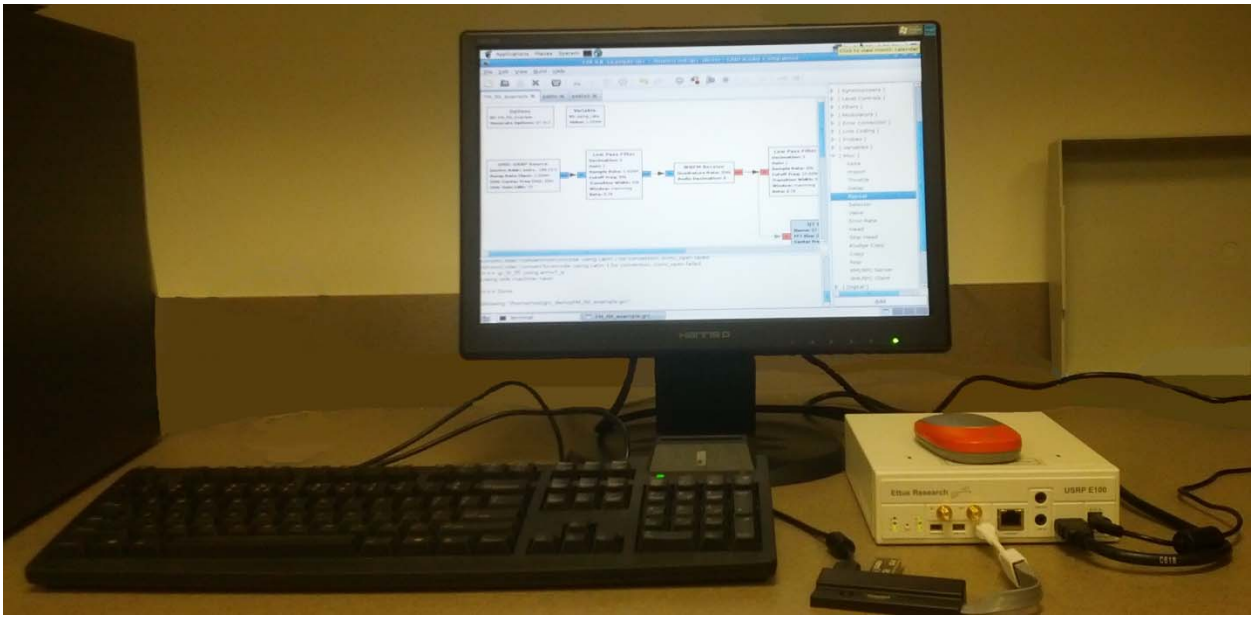

Fig. 3 Overview of E100 Set-up for Stand-alone Operation

Fig. 4 Close-up of E100 in Stand-alone Operation

There are two different methods to operate the E100 from another machine. The first is to use a standard-A to mini-B USB cable from the "console" port on the E100 to any USB port on the second machine. This approach requires the GNU Screen application to be installed on the host machine – "sudo apt-get install screen" will install the application. Fig. 5 shows the E100 connected to work with the "screen" command. Run "dmesg" in a terminal window on the second machine to locate which USB port is connected to the USRP. If you have just connected the USRP to the host machine, then fig. 6 shows what you should see near the bottom of the terminal output.

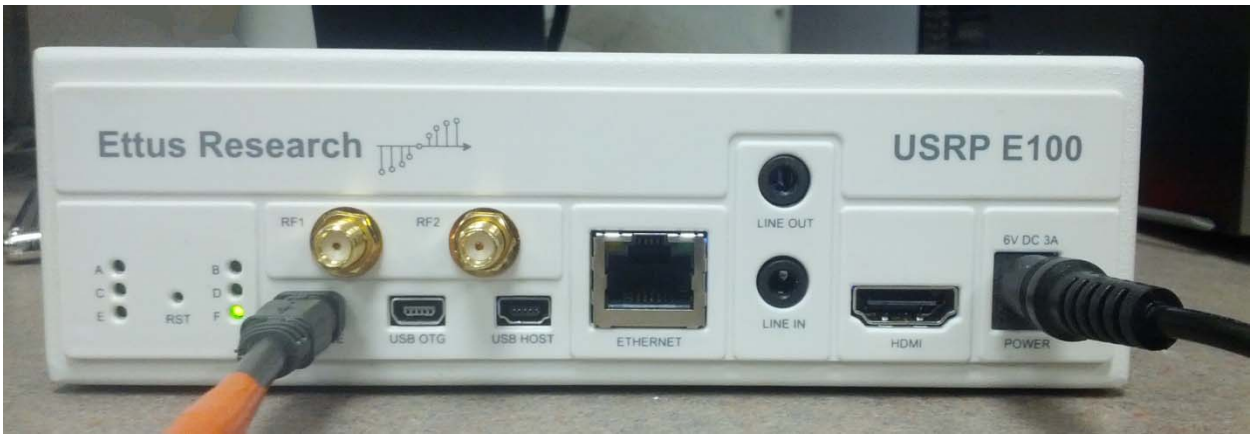

Fig. 5 Close-up of E100 Set-up for Screen Mode Operation

```
[1318861.080013] usb 5-1: >new full-speed USB device number 2 using uhci_hcd
[1318861.338032] usb 5-1: >New USB device found, idVendor=0403, idProduct=6001
[1318861.338036] usb 5-1: >New USB device strings: Mfr=1, Product=2, SerialNumber=3
[1318861.338039] usb 5-1: >Product: FT232R USB UART
[1318861.338041] usb 5-1: >Manufacturer: FTDI
[1318861.338043] usb 5-1: >SerialNumber: AE00EE7Z
[1318861.959095] usbcore: registered new interface driver usbserial
[1318861.959112] usbcore: registered new interface driver usbserial_generic
[1318861.959123] USB Serial support registered for generic
[1318861.959129] usbserial: USB Serial Driver core
[1318861.971100] usbcore: registered new interface driver ftdi_sio
[1318861.971111] USB Serial support registered for FTDI USB Serial Device
[1318861.971193] usb 5-1: >Detected FT232RL
[1318861.971195] usb 5-1: >Number of endpoints 2
[1318861.971197] usb 5-1: >Endpoint 1 MaxPacketSize 64
[1318861.971198] usb 5-1: >Endpoint 2 MaxPacketSize 64
[1318861.971200] usb 5-1: >Setting MaxPacketSize 64
[1318861.974087] usb 5-1: >FTDI USB Serial Device converter now attached to ttyUSB0
[1318861.974101] ftdi_sio: v1.6.0:USB FTDI Serial Converters Driver
```

Fig. 6 Terminal Window Output of "dmesg" Command

The line "…now attached to ttyUSB0" (second from the bottom) is important because "ttyUSB0" is the address of the USRP on the host machine. That address is then used to create a screen on the host machine by using the following command:

```
sudo screen /dev/ttyUSB0 115200,cs8,-ixon,-ixoff
```

Once this command has been entered, power on the E100 and the system should begin to boot. Fig. 7 shows the result of a successful boot and the log-in screen. Again, the default username is "root" and the default password "usrpe".

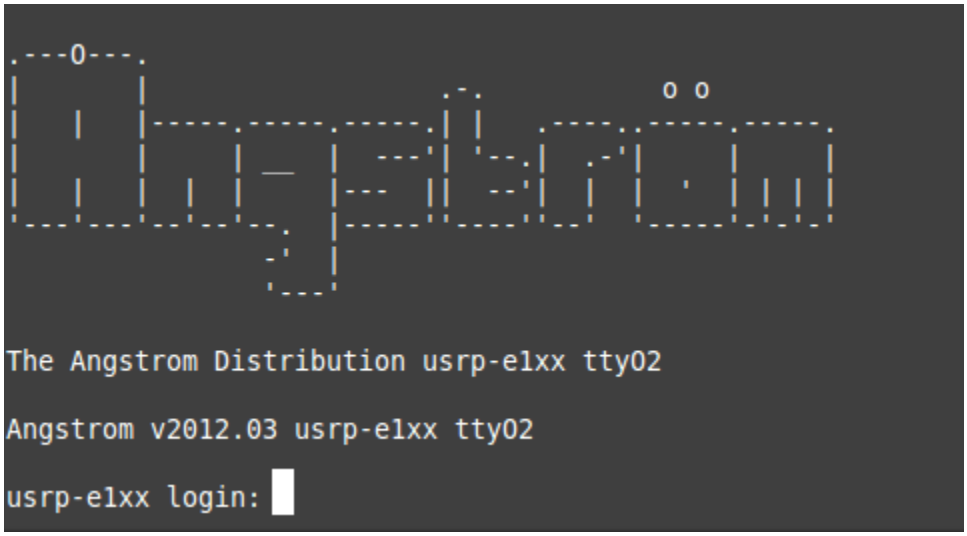

Fig. 7 Screen Mode Operation: Log-in Menu

The second method uses the network port on the E100 similar to previous USRP versions. An Ethernet cable must be connected into the Ethernet port on the E100 and the host machine. If you

know the IP address of the E100, then you can use the “ssh” command to tunnel into the device and proceed similar to the screen approach. By default, the E100 is not set with a static IP address. This means that if your E100 is straight from the factory, you may need to set the IP address using one of the other two methods before you can communicate through the network port. When you have vented your frustration and applied either the stand-alone or screen approach, open a terminal window and type the following command:

```
Vi /etc/network/interfaces
```

The command will open a virtual text editor where the static IP address for the USRP can be set. Fig. 8 shows what should appear in the virtual text editor; use the arrow keys to scroll through the document until you find the segment seen in fig. 9.

```
# /etc/network/interfaces -- configuration file for ifup(8), ifdown(8)

# The loopback interface
auto lo
iface lo inet loopback


# Wireless interfaces
#
# Example of an unencrypted (no WEP or WPA) wireless connection
# that connects to any available access point:
#
iface wlan0 inet dhcp
        wireless_mode managed
        wireless_essid any
#
#
# Same as above but locked to a specific access point:
#
#iface wlan0 inet dhcp
#       wireless_mode managed
#       wireless-essid some-essid
#
                                                          1,28          Top
```

Fig. 8 Virtual Editor: Network Interfaces Configuration

```
# Wired or wireless interfaces
auto eth0
```

Fig. 9 Virtual Editor: Network Interfaces Configuration – Wired Network Connections before Editing

Move the cursor to the end of the line that ends with “`eth0 inet dhcp`” and type “`X`”. This will delete text one character back from the cursor. Delete the word “dhcp”. Once this is done, type “`a`” to begin inserting text and add the lines seen in fig. 10. After all text has been entered, hit “Esc” to exit editing mode and then type “`:X`” to save and exit the virtual editor.

```
# Wired or wireless interfaces
auto eth0
iface eth0 inet static
        address 192.168.10.2
        network 192.168.0.0
        netmask 255.255.255.0
        broadcast 192.168.10.255
        gateway 192.168.10.2
```

Fig. 10 Virtual Editor: Network Interfaces Configuration – Wired Network Connections after Editing

After editing the network interfaces, type "`sudo ifdown eth0`" and then "`sudo ifup eth0`" to make the changes take effect. The purpose of editing network interfaces is to set the static IP address as part of the booting process. This saves the user from having to set the static IP address every time they power on the USRP. Once the static IP address has been set, the USRP can be accessed through the network port with the "ssh" command. Fig. 11 shows the E100 set-up to communicate via the network port.

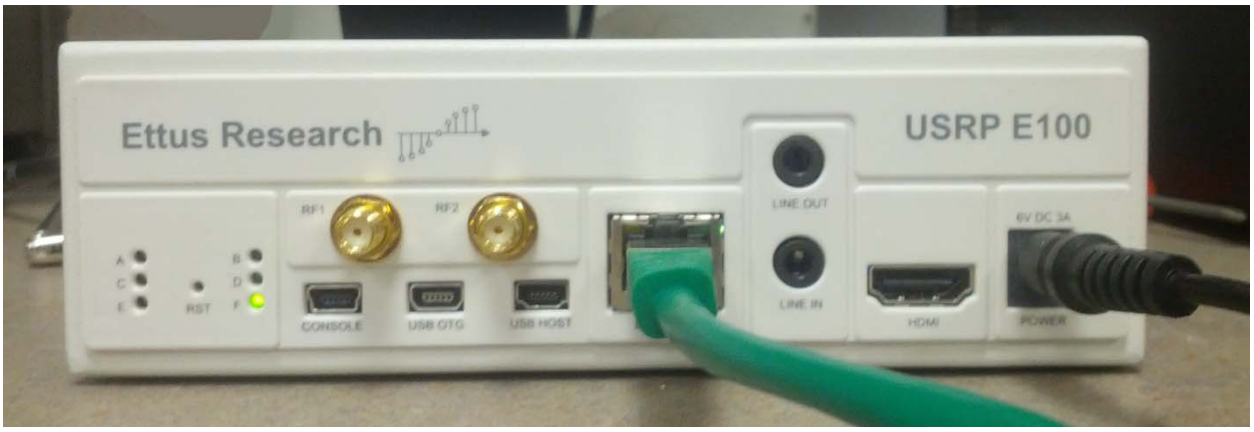

Fig. 11 E100 in Network Operations Mode

Can you step through an example in GNU Radio?
Now that you can communicate with the E100, turn on the device in stand-alone mode. Once you are logged in, open a terminal and type "gnu-radio companion". This will open the GNU Radio Companion GUI where you can create anything your heart desires. An easy project to start with is an FM receiver. Due to differences in the hardware between the E100 and the USRP2, this FM receiver will be different from the one shown in the Ettus Research YouTube video.

In the GNU Radio Companion, double click on the "Options" block to edit the properties of that block. Change the project name in the "ID" field to "FM_RX_Example" and under generate GUI, select "none". Now save the project as "FM_RX_Example.grc". The name of the project must match the file name where it is saved. Edit the "Variable" block so that the "ID" is "samp_rate" and the value is "1.024e6".

To add a block, you can search through the available blocks or use the "CTL + f" key combo to search for a particular block. Find the "UHD: USRP Source" block (it is under sources) and add it to the flow graph. Edit the sample rate field to "samp_rate" (the variable that was just set), the

center frequency to 25 MHz and the gain to 70 dB.

Next, add a low pass filter block and connect it to the “USRP Source” block by clicking on the out box on the “USRP Source” and the in box on the low pass filter. The head of the arrow is black, which means that the type leaving the “USRP Source” is the same type required by the input to the low pass filter. If the arrow head is red, then there is a type mismatch. Edit the low pass filter block so that decimation is 4, gain is 1 dB, sample rate is equal to “samp_rate”, the cutoff frequency is 95 kHz and the transition width is 45 kHz. The purpose of this filter is to take advantage of aliasing to bring the signal down to frequencies at which the USRP can operate. This same concept is utilized in the “USRP Source” – since the sample rate is 1.024 MHz, the signal is aliased down to 512 kHz (the Nyquist rate for a signal at 512 kHz is 2*512kHz which equals 1.024 MHz). In the low pass filter, the signal is aliased further down to 128 kHz.

After the low pass filter will be a “WBFM Receive” block. This stands for wideband frequency modulation. Connect this block to the low pass filter and edit the quadrature rate to be 256 kHz and the audio decimation rate to be 8.

Following the “WBFM Receive” will be another low pass filter. The function of this low pass filter is to alias the signal and to remove noise from the transmission. Connect the output of the “WBFM Receive” block to the second low pass filter. Now, edit the low pass filter so that the decimation rate is 2, the gain is 1 dB, the sample rate is 32 kHz, the cutoff frequency is 10.625 kHz and the transmission width is 5 kHz.

Finally, add an audio sink block to the flow graph. Connect the output of the second low pass filter to the input to the audio sink. Edit the audio sink block so that the sample rate is 16 kHz and the device is “`hw:0,0`”. This will allow the USRP to play the audio generated by the FM receiver out of the “line out” port. Fig. 12 shows the final flow graph in GNU Radio Companion.

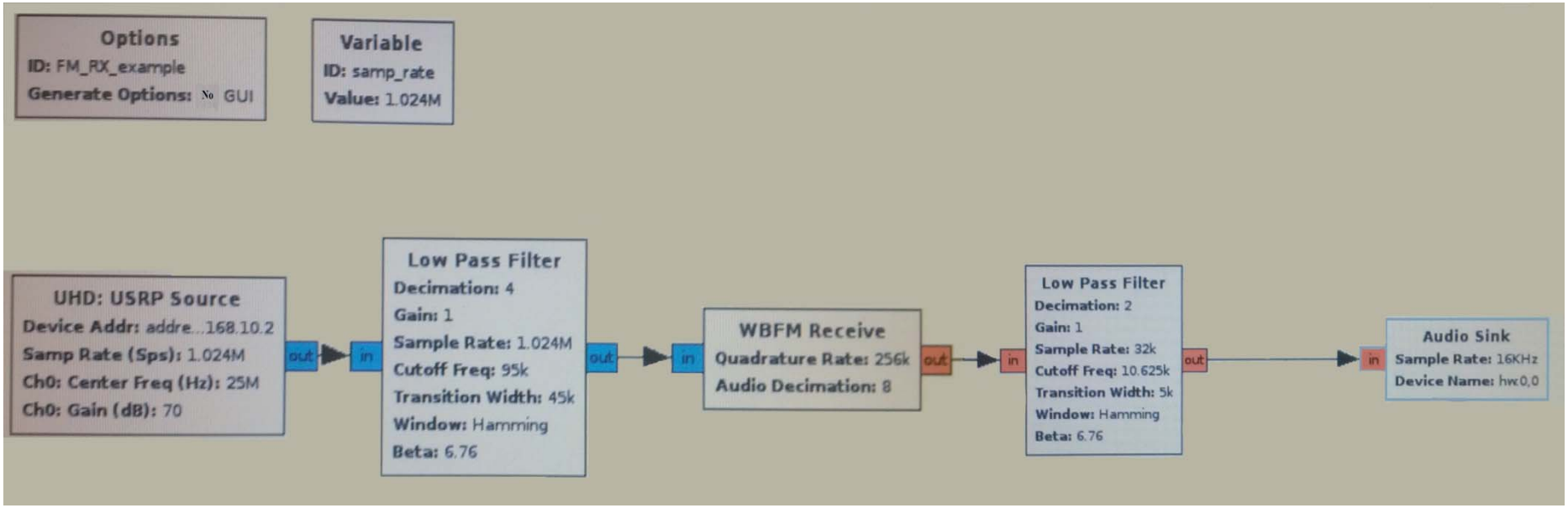

Fig. 12 Completed FM Receiver Flow Graph

Appendix B: MATLAB PSK31 Transmitter Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Task: Take Text Stream and create PSK-31 Signal       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc, clear, close all

%enter the message as a string
message=input('Write Message Here: ', 's');

L = length(message);

code = [];
base = [];

for k = 1:L
    y = varicode_lookup(message(k));
    code = [code y 0 0];
end

code = [0 code];

for i = 1:8
    code = [0 0 0 0 0 0 0 0 0 0 code];
end

%Create the alternating base waveform (series of 0 bits)
for k = 1:length(code)
    base=[base mod(k-1,2)];
end

netshift(1) = ~base(1);
%Combine the base 01010... pattern with the code to produce PSK31
for k = 2:length(base)
    if code(k) == 1
        netshift(k) = netshift(k-1);
    else
        netshift(k) = ~netshift(k-1);
    end
end



%%%%PSK-31 Signal Constants
fc = 31.25;
f_carrier = 1e3;
Tc = 1/fc;
fs = 16e3;
Ts = 1/fs;
PSK_amp = 0.65;

%%%%length of time vector will be determined by the number of bits
num_bits = length(netshift);
T = 0:Ts:((num_bits*fs/fc-1)*Ts);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Expand the sim_code vector to accomodate for sampling rate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

newbit      = [];
PSK31_base  = [];

%Expand the warble vector
for i=1:length(netshift)
    for j=1:(fs/fc);
        newbit = [newbit netshift(i)];
    end

    PSK31_base = [PSK31_base newbit];
    newbit     = [];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Filter the Baseband bits
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure(1)
plot(T, PSK31_base,'g')
grid on
% for j=1:num_bits+81
%     line([j*(1/fc) j*(1/fc)], [-3 3])
% end
xlabel('Time (Seconds)'), ylabel('Amplitude (Volts)')
title('PSK31 Baseband Bits')
axis([0 (((num_bits)*fs/fc-1)*Ts) -1 2]);

[freq1, power1] = get_spectrum(PSK31_base, fs);

figure(2)
plot(freq1, power1)
grid on
xlabel('Frequency (Hz)'), ylabel('Power (dB)')
title(['PSK31 Baseband Bits: Frequency Spectrum, f_{s} = ' num2str(fs)])
axis([-100 100 -60 5]);


%LPF is actually done here
PSK31_filtered = filter_Baseband(PSK31_base, 15, fs);



figure(3)
plot(T, PSK31_filtered,'g');
for j=1:num_bits
    line([j*(1/fc) j*(1/fc)], [-3 3])
end
xlabel('Time'), ylabel('Amplitude')
title('PSK31 baseband')
%axis([0 (((num_bits+81)*fs/fc-1)*Ts) -3 3]);
```

```
[freq2, power2] = get_spectrum(PSK31_filtered, fs);

figure(4)
plot(freq2, power2)
grid on
xlabel('Frequency (Hz)'), ylabel('Power (dB)')
title(['PSK31 Baseband Bits after LPF: Frequency Spectrum, f_{s} = ' num2str(fs)])
axis([-100 100 -60 5]);

figure(5)
subplot(2,1,1)
plot(freq1, power1)
grid on
xlabel('Frequency (Hz)'), ylabel('Power (dB)')
title(['PSK31 Baseband Bits: Frequency Spectrum, f_{s} = ' num2str(fs)])
axis([-100 100 -60 5]);
subplot(2,1,2)
plot(freq2, power2)
grid on
xlabel('Frequency (Hz)'), ylabel('Power (dB)')
title(['PSK31 Baseband Bits after LPF: Frequency Spectrum, f_{s} = ' num2str(fs)])
axis([-100 100 -60 5]);

%%%Create the PSK-31 waveform
PSK31_modulated = cos(2*pi*f_carrier*T) .* PSK31_filtered;



figure(6)
plot(T, PSK31_modulated,'g');
for j=1:num_bits
    line([j*(1/fc) j*(1/fc)], [-3 3])
end
xlabel('Time'), ylabel('Amplitude')
title('PSK31 @ 1kHz')
%axis([0 (((num_bits+81)*fs/fc-1)*Ts) -3 3]);


figure(7)
subplot(2,1,1)
plot(T, PSK31_filtered,'g')
grid on
for j=1:num_bits
    line([j*(1/fc) j*(1/fc)], [-3 3])
end
xlabel('Time'), ylabel('Amplitude')
title('PSK31 baseband')
axis([((75)*fs/fc-1)*Ts) (((num_bits)*fs/fc-1)*Ts) -1 1]);
subplot(2,1,2)
plot(T, PSK31_modulated,'g')
grid on
for j=1:num_bits
    line([j*(1/fc) j*(1/fc)], [-3 3])
end
xlabel('Time'), ylabel('Amplitude')
title('PSK31 @ 1kHz')
axis([((75)*fs/fc-1)*Ts) (((num_bits)*fs/fc-1)*Ts) -1 1]);
```

```
[freq3, power3] = get_spectrum(PSK31_modulated, fs);

figure(8)
plot(freq3/1000, power3)
xlabel('Frequency (kHz)'), ylabel('Power (dB)')
title(['PSK31 @ 1kHz: Frequency Spectrum, f_{s} = ' num2str(fs)])
axis([-2.5 2.5 -60 5]);



%%%Add the 0.750 seconds of unmodulated carrier to the end of the signal
T2 = (((num_bits)*fs/fc-1)*Ts):Ts:((((num_bits)*fs/fc-1)*Ts)*Ts+0.750);
T = [T T2];
trail = cos(2*pi*f_carrier*T2);
net_signal = [PSK31_modulated trail];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Filter the shifted copies
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

PSK31_final = filter_IF(net_signal, 1000-15, 1000+15, 16e3);
%PSK31_final = net_signal;

[freq4, power4] = get_spectrum(PSK31_final, fs);

figure(9)
plot(freq4/1000, power4)
xlabel('Frequency (kHz)'), ylabel('Power (dB)')
title('PSK31 Filtered: Frequency Spectrum, f_{c} = 1000')
axis([-2 2 -60 5]);


%%%Write a .wav file with the PSK31_final
audiowrite('NewHopePSK@1k.wav',PSK31_final,fs);
```

Appendix C: MATLAB PSK31 Receiver Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Task: Take PSK-31 Signal and recreate the message     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear, clc, close all

%enter the message as a string
f_carrier=input('What is the center frequency of the signal? ');

%load the PSK31 wav file
PSK31_TX = transpose(wavread('NewHopePSK@1k.wav'));

%Back out the time vector
fs = 16e3;
Ts = 1/fs;
num_samples = length(PSK31_TX);
amount_time = num_samples / fs;

t = 0:Ts:amount_time-Ts;

%MIX the PSK31 Signal with a cosine wave at the carrier frequency
base = cos(2*pi*f_carrier.*t);

figure(1)
subplot(2,1,1)
plot(t, base)
xlabel('Time (sec)'), ylabel('Amplitude (v)')
title('Carrier Waveform')
subplot(2,1,2)
plot(t, PSK31_TX)
xlabel('Time (sec)'), ylabel('Amplitude (v)')
title('PSK31 Waveform')

mixed = base .* PSK31_TX;

figure(2)
plot(t, mixed)
xlabel('Time (sec)'), ylabel('Amplitude (v)')
title('Mixed Signal')


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Integrate over a bit period
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

step2 = [];
for i = 1:length(mixed)/(fs/31.25)
    temp = [];
    for j = 1:512
        temp(j) = mixed((i-1)*512+j);
    end
    step2 = [step2 intdump(temp, 512)];
end
```

```
%Multiply area by the bit period
step2 = step2/31.25;

figure(3)
plot(step2)
ylabel('Energy')
title('Samples Integrated by Bit Period (0.032 seconds)')

%use the sign of the integrated samples to determine bits
phase_value = (sign(step2)+1)/2;

%plot the bits
figure(4)
plot(phase_value, 'bo'), grid on
xlabel('Time (sec)'), ylabel('Amplitude (v)')
title('Phase of the Signal')
axis([0 length(phase_value)+5 -0.5 1.5])


%Pick out the varicode
varicode_bits = [];
for i = 2:length(phase_value)
    varicode_bits(i-1) = ~bitxor(phase_value(i),phase_value(i-1));
end

varicode_bits = [varicode_bits 0];

%plot the varicode
figure(6)
plot((Ts:Ts:(num_samples/512)*Ts), varicode_bits, 'bo'), grid on
xlabel('Time (sec)'), ylabel('Amplitude (v)')
title('Baseband Bits')
axis([0 (num_samples/512+1)*Ts -0.5 1.5])

%Convert the varicode back into text
start_index  = [];
finish_index = [];

for i = 3:length(varicode_bits)
   %Find all the starts to varicode sequences
   if varicode_bits(i-2) == 0 && varicode_bits(i-1) == 0
        if varicode_bits(i) == 1
            start_index = [start_index i];
        end
   end

   %Find all the ends to varicode sequences
   if (varicode_bits(i-2) == 1) && (varicode_bits(i-1) == 0)
        if varicode_bits(i) == 0
            finish_index = [finish_index i-2];
        end
   end
end

%Look-up the varicode sequences to get the chars
message = [];
varicode_sequence = [];
```

```
for i = 1:length(start_index)
    for j = start_index(i):finish_index(i)
        varicode_sequence = [varicode_sequence num2str(varicode_bits(j))];
    end

    message = [message array_to_text_s(varicode_sequence)];
    varicode_sequence = [];
end

message
```

Appendix D: Doppler Curve Plots from ISS Data

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Plot Doppler curves of ISS                                                %%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

t1 = 0:10:240;
a1 = [319, 320, 322, 323, 324, 326, 327.8, 329.8, 332.2, 334.5, 339, 341, 343, 350,
353, 357, 1, 6, 10.6, 15.6, 21, 26, 35, 41, 45];
e1 = [2.8, 3.9, 4.9, 6.0, 6.9, 7.8, 8.9, 10.1, 11.4, 12.6, 14, 15, 17, 18, 19.7,
20.7, 22, 23, 23, 23.6, 24, 24, 24, 23, 22];
ds1= [9146, 9048, 8923, 8785, 8652, 8528, 8349, 8109, 7814, 7506, 7126, 6780, 6280,
5550, 4888, 4330, 3503, 2890, 2200, 1300, 650, -300, -1556, -2550, -3190];

figure(1)
plot(t1, ds1)
xlabel('Time (sec)')
ylabel('Dopple Shift (Hz)')
title('Dopple Curve for ISS: Some Skirting Pass')


t2 = 0:15:(19*15);
a2 = [243, 244, 247, 250, 255, 265, 283, 319, 358, 18, 29, 34, 38, 41, 42, 44, 45,
46, 46, 47];
e2 = [26, 30, 35, 41, 49, 58, 67, 71, 65, 57, 48, 41, 34, 29, 25, 22, 19, 16, 14,
12];
ds2= [8862, 8438, 7857, 7003, 5892, 4093, 2022, -700, -2993, -4880, -6330, -7440, -
8113, -8668, -8987, -9230, -9450, -9600, -9700, -9781];

figure(2)
plot(t2, ds2)
xlabel('Time (sec)')
ylabel('Dopple Shift (Hz)')
title('Dopple Curve for ISS: Overhead Pass')


t3 = 0:15:(15*25);
a3 = [184, 182.4, 181, 179, 177, 175, 173, 170.4, 167, 164.7, 162, 158, 154, 151,
147, 142.7, 138.5, 134, 129, 125, 120.2, 115, 111.7, 107, 103.4, 99.7];
e3 = [-0.5, 0.3, 1, 1.7, 2.5, 3.3, 4.2, 5.0, 5.8, 6.6, 7.4, 8.1, 8.8, 9.5, 10.1,
10.5, 10.9, 11.1, 11.2, 11.2, 11.0, 10.7, 10.2, 9.7, 9.1, 8.4];
ds3= [8220, 8070, 7907, 7708, 7476, 7210, 6930, 6624, 6243, 5819, 5390, 4872, 4390,
3600, 2998, 2303, 1505, 750, 90, -725, -1556, -2270, -2980, -3730, -4260, -4830];

figure(3)
plot(t3, ds3)
xlabel('Time (sec)')
ylabel('Dopple Shift (Hz)')
title('Dopple Curve for ISS: Large Skirting Pass')


t4 = 0:15:(15*32);
```

```
a4 = [278, 277, 275, 273, 211, 269, 267.5, 265.5, 263.4, 261.2, 258.9, 256.7,
254.4, 252.0, 249.6, 247.3, 244.8, 211.9, 239.6, 237.0, 234.6, 232.5, 230, 227.7,
255.5, 223.2, 220.9, 218.8, 216.6, 214, 212.5, 210.6, 208.7];
e4 = [-6.5, -6, -5.6, -5.2, -4.9, -4.6, -4.2, -3.9, -3.6, -3.3, -3.1, -2.9, -2.7, -
2.5, -2.4, -2.3, -2.2, -2.1, -2.1, -2.2, -2.3, -2.4, -2.5, -2.7, -2.8, -3.1, -3.3,
-3.6, -3.8, -4.2, -4.5, -4.9, -5.3];
ds4= [5880, 5630, 5390, 5140, 4890, 4600, 4330, 4010, 3680, 3350, 3010, 2640, 2290,
1880, 1490, 1080, 680, 200, -150, -540, -940, -1330, -1720, -2130, -2490, -2860, -
3210, -3540, -3870, -4200, -4470, -4750, -5010];

figure(4)
plot(t4, ds4)
xlabel('Time (sec)')
ylabel('Dopple Shift (Hz)')
title('Dopple Curve for ISS: Tangential Pass')
```

Appendix E: Doppler Curve Generator MATLAB Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%    Doppler Curve Generator: Based on ISS Data                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc, clear

%ISS Data - simplified
orbit_height = 418.5; %Average Height of ISS in km (apogee+perigee)/2
orbit_speed  = 7.66;  %Average Speed of ISS is km/sec
freq_station = 437.8e6; %Operating Frequency of ISS [Hz]

Ts = 1;   %Sample Period [seconds]
Time_end = 745 - mod(745,Ts); %Max possible time(longest overhead case)
t = 0:Ts:Time_end; %time vector in seconds


%Generate Start Points for the Pass
prompt = 'Choose the starting coordinates of the pass. The astronomical horizon is
3000 km in diameter, so choose an initial condition between -1500 km and 1500 km.'
y0 = input('Input the starting condition (km): ');
y1 = input('Input the ending   condition (km): ');
x0 = -2000; % [km]
x1 =  2000; % [km]

%Calculate the Pass Line
x = -2000:4000/length(t):2000; % [km]
m = (y1 - y0)/4000;
b = 2000*m+y0;

PassLine = m.*x+b; % [km]


%Generate the Astronmical Horizon
x2 = -1500:3000/length(t):1500; % [km]
horizon_top    = sqrt(1500^2 - x2.^2);
horizon_bottom = -1*sqrt(1500^2 - x2.^2);


%Plot the overhead view of the Satellite Pass
figure (1)
plot(x,PassLine, 'b ', x2, horizon_top, 'r', x2, horizon_bottom, 'r'), grid on
xlabel('Distance in x-direction relative to RX (km)'), ylabel('Distance in y-
direction relative to RX (km')
title('Overhead View of Satellite Pass - Flat-World Assumption')
axis([-2500, 2500, -2000, 2000])

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Convert the Satellite Pass Equations to time-based functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

d_sec        = orbit_speed;               %time covered in one second by the satel-
lite
d_total     = sqrt((x1-x0)^2+(y1-y0)^2);  %total distance covered for this overhead
pass
```

```
time_needed = d_total / d_sec;           %amnt of time needed to cover the total
distance
delta_x     = (x1-x0)/time_needed;       %step-size for the x-variable

x_t = delta_x*t + x0;
y_t = m.*x_t + b;
z_t = orbit_height;


%Check that the time-based functions are correct
figure(2)

subplot(2,1,1)
plot(x,PassLine, 'b '), grid on
title('Check of Time-Based Functions - Flat-World Assumption')
xlabel('x-distance relative to RX (km)'), ylabel('y-distance relative to RX (km)')
axis([-2500, 2500, -2000, 2000])

subplot(2,1,2)
plot(x_t,y_t, 'r '), grid on
xlabel('x-distance relative to RX (km)'), ylabel('y-distance relative to RX (km)')
axis([-2500, 2500, -2000, 2000])


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%   Convert Cartesian Coordinates into Spherical Coordinates          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

r_t     = sqrt(x_t.^2 + y_t.^2 + z_t.^2);    %result in km
theta_t = atan(y_t./x_t);                    %result in radians
phi_t   = acos(z_t./r_t);                    %result in radians


figure(3)

subplot(3,1,1)
plot(t,r_t), grid on

subplot(3,1,2)
plot(t,theta_t), grid on

subplot(3,1,3)
plot(t,phi_t), grid on

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%   Calculate the Doppler Shift in Two Directions                     %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Z-direction will not cause Doppler shift, but the x and y changes will
%in spherical coordinates

%Create a t2 vector and delta_v_t vector
t2        = (Ts/2):Ts:Time_end-(Ts/2); %time vector in seconds
delta_v_t = 1:length(t2);
DS_t      = 1:length(t2);
```

```
for i = 1:length(t)-1
    delta_v_t(i) = -(r_t(i+1)-r_t(i))/Ts;  %result in km/sec
    DS_t(i)      = freq_station*delta_v_t(i)*1000/(2.99e8);
end

figure(4)
plot(t2, DS_t/1000), grid on
xlabel('Time (sec)'), ylabel('Doppler Shift (kHz)')
title('Doppler Shift vs. Time')
%axis([0 100 -10 10])
```